This is to certify that the

dissertation entitled

VERIFICATION OF MOS CIRCUITS AT THE SWITCH LEVEL

presented by

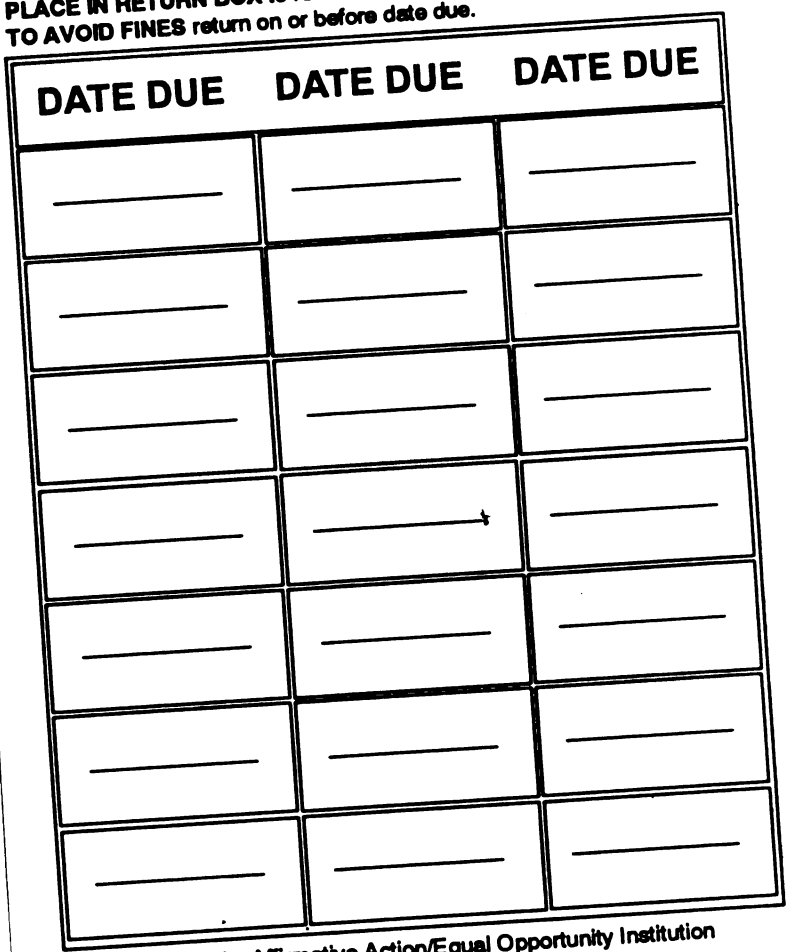
JIANN LIAO

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Compter Science

Major professor

Date 01/08/1990



VERIFICATION OF MOS CIRCUITS

AT THE SWITCH LEVEL

By

Jiann Liao

A DISSERTATION

Submitted to

Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1990

ABSTRACT

VERIFICATION OF MOS CIRCUITS AT THE SWITCH LEVEL

By

Jiann Liao

Because of the nature of MOS transistors, digital MOS circuits have been modeled at the switch level rather than the traditional gate level. In this research, a novel switch level circuit representation called *Structured LogIcal Circuit Expression (SLICE)* is proposed to represent both the connectivity and signal flow information of MOS circuits. Effective circuit verification methods based on the proposed representation are presented. Circuit verification is crucial in today's VLSI circuit design to ensure the correctness of the design. The proposed switch level verification methods include functionality extraction, logic simulation, and signal flow analysis in timing verification. The first two methods are used to assure the functional correctness of the design, while the third is used to examine the timing constraints of the design.

When performing functional verification, the functionality (or Boolean behavior) of a design must be extracted so that the functionality comparison between a design and its specification can be performed. Several rules are developed from the SLICE representation to do the functionality extraction of different MOS circuit design styles. These rules are capable of verifying both the functional correctness and the electrical safety of a design. The whole functionality extraction process is achieved by a divide-and-conquer algorithm which guides the application of the rules. Since SLICE describes the functional as well as structural information contained in the circuits, efficient switch level logic simulation can be demonstrated on the circuits represented by the proposed

representation. Experiment results are presented for functionality extraction.

To investigate the timing behavior of circuits, we propose a switch level timing verification method based on signal flow analysis of circuits in SLICE representation. Due to a lack of proper functional information, signal paths may be falsely reported by timing verifiers for some circuits. An algorithm is proposed to eliminate the false signal paths from the possible paths. Furthermore, a heuristic algorithm is presented to expedite the signal pathfinding process.

In conclusion, the contribution of this work to the verification of MOS circuits is summarized and suggestions are offered for future research.

To my parents

ACKNOWLEDGEMENTS

I wish to thank my advisor, Dr. Lionel M. Ni, for his guidance and encouragement over the years of research work leading to this dissertation. I am also indebted to Dr. Anthony S. Wojcik, Dr. Sakti Pramanik, and Dr. Dorian Feldman for their many valuable suggestions and comments on this work.

I would also like to acknowledge my fellow graduate students who have helped me in the course of my studies for the Ph.D.: S.W. Chen, E. Wu, C.T. King, C.J. Lee, D. Ra, D. Vineyard and C.J. Yang. A word of thanks is also due to Mr. M. Smith for editing the final manuscript of the dissertation.

Finally I would like to thank my family for their love and their constant support of my graduate studies.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

In today's VLSI (Very Large Scale Integrated) circuit design, MOS (Metal Oxide Silicon) technology [MeCo80, WeEs85] is the dominant technology used. It is possible now that hundreds of thousands of MOS devices may all be fabricated in a single chip. To deal with this huge amount of devices, VLSI verification tools are indispensable to the success of a design. Circuit verification is an important step in the process of ensuring the correctness of the circuit design. Several kinds of verification may be required for different verification purposes and different degrees of verification confidence. Commonly used verification methods include functional verification, timing verification, and logic simulation.

Due to the nature of MOS transistors, digital MOS circuits have been modeled at the switch level [Brya84, Haye82], instead of the traditional gate level. Therefore, many MOS circuits are often designed and verified at the switch level. Identification of the signal flow direction of each transistor in the design is essential for performing different kinds of verification. Only when all the correct signal paths are determined can timing verification be performed correctly [Joup87a, Oust85]. Knowing the signal flow direction of every transistor in the circuit also facilitate efficient switch level simulation [RaTr87,

VaDS85]. Furthermore, until the correct signal path direction of the circuit is known, functional verification cannot be properly applied to this circuit [WuNW87]. When only the circuit connectivity information is available in a design, it is difficult to derive the correct signal flow directions of the transistors in the circuit [Joup87b, Oust85]. However, a designer has the full knowledge of his/her own design, and can thus specify the signal flow directions of the transistors in the design. Therefore, one proper way to verify the design of a MOS circuit is as follows.

(1) Pre-verify the circuit design, using estimated parasitics and known signal flow directions specified by the designer before performing circuit layout;

(2) Compare precise parasitics, obtained by a circuit extractor after the circuit layout is done, with the estimated parasitics used in the pre-verification and adjust the pre-verification result; and

(3) Use wirelist comparison [Wata83, EbZa83, KoMc86, OTOO86] to make sure that the layout-extracted wirelist matches the correct wirelist which has been pre-verified in step (1).

From the above steps, we know that a circuit representation is demanded which can represent both the signal flow and interconnection of a design. Since circuit schematic representation only carries structural information, it cannot embed signal flow information in the representation. Therefore, other circuit design representations are required to carry signal flow information in addition to structural information.

The *Logical Circuit Expression* proposed in [WuNW87] allows symbolic verification of the functional correctness of MOS circuits. However, it cannot completely determine the circuit connectivity and does not describe the signal flow as specified by the designer. Thus, it cannot be used as a design representation. A MOS circuit representation called *Structured LogIcal Circuit Expression* (SLICE) is proposed for this purpose. This representation is an extension of the Logical Circuit Expression

and eliminates its aforementioned shortcomings. The SLICE unifies the representation of two major logic design styles, gate logic and path transistor logic, in MOS circuit design. The SLICE also provides a systematic way to describe both the signal flow direction and circuit connectivity of a design. Therefore, the SLICE representation can properly facilitate those processes of circuit verification which require signal flow information such as timing verification, logic simulation, and functional verification. Circuit verification can be performed more efficiently when it is based on the SLICE representation because extra information is provided in addition to circuit connectivity information. The key point in the efficient verification of a circuit is the expression of the circuit in terms of a good representation.

It is also desired to automatically convert a circuit schematic, which only carries connectivity information, into a circuit representation carrying functional information in addition to structural information. By this automated process, the designer's mistakes in specifying signal flow can be eliminated, and the confidence of the design can be increased. However, it turns out that this can be done only for most parts of a circuit. Some part of the circuit will still need human intervention to complete the whole conversion [Joup87b, Oust85]. Therefore, new methods to convert the representation of a circuit schematic into the desired representation are necessary.

The objective of this research is to develop a MOS circuit representation, namely SLICE representation, which is able to provide both the functional (or signal flow) and structural (or connectivity) information of a circuit. Efficient circuit verification methods based on the proposed representation will also be developed.

1.2 Background

This section gives the background in switch level model of MOS circuits and circuit verification methods of VLSI design. The need to have one level lower from the traditional gate level down to the switch level in digital MOS circuit design is explained.

Three verification methods emphasized at the switch level are introduced: logic simulation, timing verification, and functional verification.

1.2.1 Switch Level Circuit Model

The use of a Boolean gate model to describe the behavior of integrated circuits consisting of MOS transistors has been widely recognized to be quite inadequate [Brya84, Haye82]. In the Boolean gate model, a circuit consists of a set of logic gates connected by unidirectional memoryless wires. The logic gates compute Boolean functions of their input signals and transmit these values along the wires to the inputs of other gates. Each gate input has a unique signal source. Information is only stored in the feedback paths of sequential circuits. Some MOS pass transistor networks, however, can implement combinational logic in ways that resemble relay contact networks more closely than conventional logic gate networks. Dynamic memories using MOS devices can store information without feedback paths by using the capacitance of the wires (interconnect region) and the gates of the transistors attached to them. A variety of bus structures can provide multidirectional, multipoint communication. Thus, MOS circuits consist of bidirectional switching elements connected by bidirectional wires. These wires contain memory due to their interconnect and device capacitances and hence, the MOS circuits cannot be modeled accurately by the Boolean gate level circuit model. Therefore, researchers have modeled the MOS circuits at the switch level [Brya84, Haye82], by treating each transistor as a perfect switch, rather than at the gate level. In the switch level model, each node in the circuit has a discrete capacitance, and each transistor in the circuit has a discrete resistance. Those discrete values determine the signal strength in the circuit.

Since VLSI circuits are mainly composed of MOS transistors, many designs have to be performed at the switch level. Traditional digital design methods at the gate level must be adjusted to handle the switch level designs. Furthermore, new methods and tools for switch level circuit design have been emerging. These switch level methods, especially

for verification purposes, have received much attention because of their successful uses in VLSI circuit design.

1.2.2 Circuit Verification

In the design of a VLSI circuit, circuit synthesis and circuit verification are two important steps in completing a successful design. Synthesis is the process of implementing the circuit based on a number of design specifications. After the circuit is designed, it is necessary to make sure that the circuit does meet the design requirements. Thus, a process called verification is required to verify the design. This verification process includes a functional correctness check and a performance constraint check. The functional correctness check is usually accomplished by logic simulation or functional verification. The performance constraint check is usually done by timing verification. We will introduce these verification methods with emphasis on switch level circuits. If these verification methods shows that a circuit does not meet the design specification, the circuit must be resigned and the verification process repeated. The iteration of the synthesis and verification will continue until the design requirements are satisfied. A diagram showing a typical VLSI design flow is given in Figure 1.1.

1.2.2.1 Switch Level Logic Simulation

Several logic simulators have been implemented based on switch level models [Brya80, BaTr80, Brya84]. These simulators are able to simulate a large variety of MOS designs, including ones containing huge transistors. The simulators accept logic values for the inputs of the circuit and observe the expected logic values at the outputs of the circuit. If the outputs are not as expected, then the design is incorrect.

Based on the switch level models, most research on switch level simulation has focused on finding efficient algorithms to compute the steady state logic values of circuits. Usually these algorithms use some form of iterative method [Brya84]. These

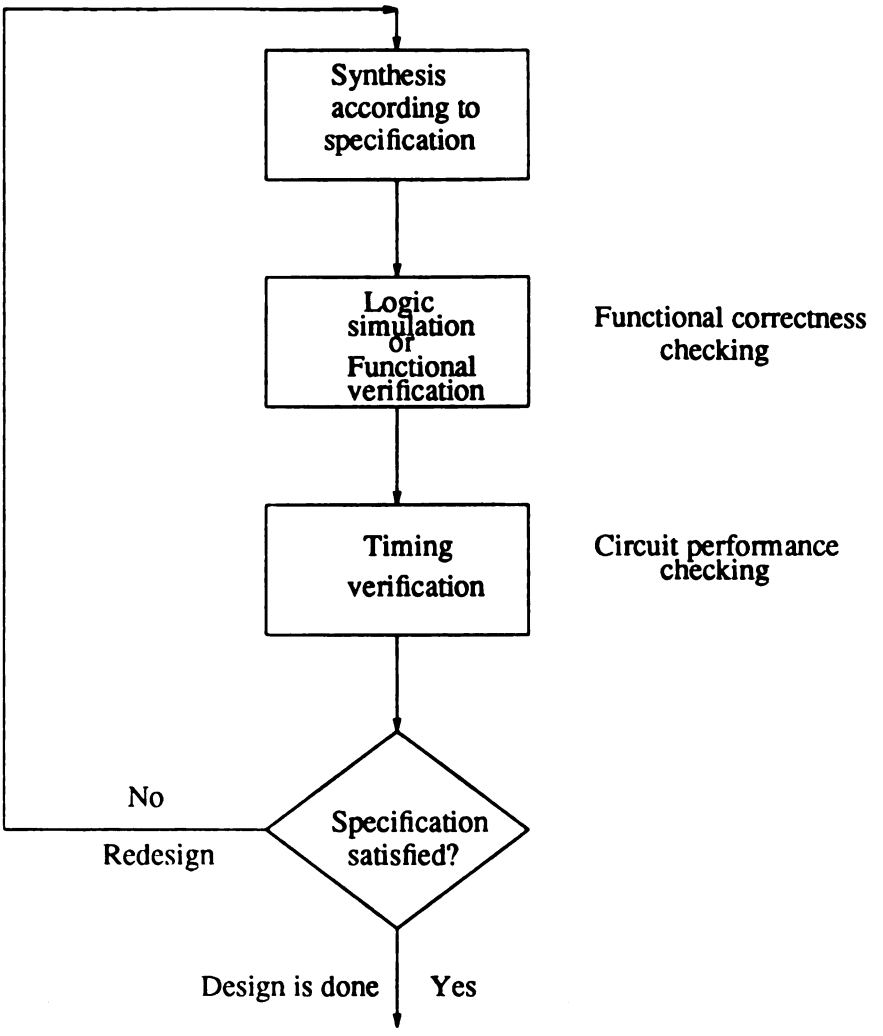


Figure 1.1 Flow of VLSI circuit design

simulators try to determine all the steady state logic values for every node in the circuit. It is not necessary to evaluate all the nodes in the circuit, however. It is sufficient to evaluate the logic values of some important nodes such as output nodes, thereby reducing the simulation time.

1.2.2.2 Timing Verification

Timing verifiers identify the longest delay path (critical path) in a circuit in order to ensure that timing requirements can be satisfied. Timing verifiers also try to improve the

circuit performance and to ensure that the clock cycles are correct [Oust85, Joup87a]. Timing verification is more like electrical-rule checking because its essential function is to traverse the circuit network. For every input and output signal, there are many possible paths in the circuit. Each path consists of a set of network nodes that connect the output of one component in the circuit to the input of another component. If the delay of each node is first determined and stored, then the verifier's job consists of recursively finding the path of worst case delay to every output. An RC tree delay model, which models the delay of circuit elements with resistance and capacitance [RuPe83], can quickly approximate most timing. To determine a path delay precisely, many timing verifiers extract the details of the longest path and then use circuit simulation to evaluate the timing fully.

There are additional considerations that must be addressed to accurately verify the delays in a circuit. In the case of MOS transistors, one such problem is their ability to function bidirectionally. This means that complex subcircuits in the circuit can present an exponential number of signal paths to the timing verifiers including many false paths. Since timing verifier operation focuses on some subset of the circuit, much help is needed to make it focus correctly. Although intelligent analysis techniques [Joup87b] are of some help, only the user knows what is correct. Thus, it may be necessary to let users specify information in the circuit such as wire direction, initial conditions, and elimination of circuitry from analysis [Oust85]. However, if a circuit representation can carry enough signal path direction information, then many false paths searched in the circuit by timing verifiers can be avoided and the time spent for timing verification can be reduced.

1.2.2.3 Functional Verification

Functional verifiers compare symbolic descriptions of circuit functionality with the derived behavior of the individual parts of the circuit, also described symbolically

[WuNW87, Brya85, Brya87b]. Although not as widely used as logic simulation, functional verification is a significant analysis technique. When both logic simulation and functional verification are successfully performed for a design, confidence in the design is increased.

Each primitive circuit component in a circuit can be described behaviorally in terms of its inputs, outputs, and internal state. By combining these component descriptions, an overall circuit behavior is derived that can symbolically represent the circuit's function. The verification consists of comparing this derived behavior with the designer-specified behavior. The two descriptions should be mathematically equivalent after comparison, in which case the circuit is successfully verified. The three steps of the functional verification process are the selection of a behavior representation, the aggregation of individual behavioral descriptions to form an overall circuit specification, and the comparison of derived functionality with specified functionality.

Usually, the circuit behavior consists of equations for the outputs and internal states of its components. Combining output and state equations is a simple matter of substitution. The resulting equations are long and complex, but they can be reduced with standard techniques. The final stage of verification compares the aggregated behavior with the designer-specified behavior. This involves showing equivalence between two sets of equations, which can be done in a number of ways.

To efficiently derive the behavior from the MOS circuit, the circuit representation itself must provide signal flow information. The derivation is especially difficult for MOS circuits because the bidirectionality and high impedance state of MOS transistors are not easy to handle. Therefore, a good circuit representation which carries signal flow information is necessary for efficient functional verification. This can be achieved by the proposed circuit representation.

1.3 Previous Work

Based on the proposed SLICE representation, this research proposes verification methods which are important in circuit verification. The first method to be presented is the functionality extraction from circuit connectivity. This method derives the Boolean expressions of a circuit from a given circuit connectivity such as circuit layout. Previous work [Brya87, WuNW87, HaSa83] has developed functionality extraction methods for switch level networks. The work of [WuNW87] does not present a general algorithm to extract the functionality of transistor groups in a MOS circuit. Nor can it handle high impedance state or do the electrical checking in the circuits.

State encoding [Brya87] and path encoding [HaSa83] approaches are used to perform functionality extraction. The method proposed in this research is a rule-based approach driven by the signal flow analysis. Since the number of rules employed is small, the proposed approach is simple and effective. When applying the proposed rules, the NP-complete Boolean equivalence problem is encountered in the functionality extraction of transistor groups. We propose the junction node concept to greatly reduce the time complexity of Boolean equivalence checking. The work in [Brya87] proposes an elegant method which has polynomial time complexity to extract the functionality of transistor groups, and does not encounter Boolean equivalence problems. However, because the state encoding is used, this extraction cannot distinguish uncertain logic states from unsafe uncertain logic states. If a logic state is an indeterminate logic level, say U, then U is usually assumed to be a safe uncertain logic state which is either logic 1 or logic 0. Then, the extraction process can still proceed. However, it is possible that U may be an unsafe uncertain logic level between logic 1 and logic 0. The proposed method is able to detect this kind of unsafe uncertain logic level, which may be caused by a design error such as a short circuit.

Another feature of our approach is that it enables electrical rule checking to be performed at the same time as the processing of functionality extraction. Thus, much work

in electrical rule checking which usually is a separate process in circuit verification can be done along with the functionality extraction process. Furthermore, the work in [Brys87, HaSa83] considers only the functionality extraction of transistor groups of a circuit, with no consideration of functionality aggregation among transistor groups. This is because they are only interested in the logic simulation of a circuit and not the functionality extraction of a circuit. At present, our approach can be applied to all static MOS circuits, but only to some dynamic circuits. More work is needed to extend it to general dynamic MOS circuits.

The next method proposed in this work addresses the problem of timing verification. It is known that in some MOS circuits timing verifiers cannot identify the correct signal flow direction in the circuit. Therefore, false critical paths may be reported [Oust85, Joup87a]. Several methods have been proposed to derive the correct signal flow direction of all the transistors in the MOS circuits. One solution requires users to tag with direction flags [Oust85] all unidirectional pass transistors whose directions are difficult to determine. Obviously, this approach involves error-prone user input. Another approach tries to derive the correct signal flow direction through a set of rules [Joup87b]. However, the sizable number of the rules complicates the application. In this work, a method to find the correct signal paths of MOS transistors in timing verification is proposed. A signal path searching scheme based on SLICE and signal direction finding for MOS transistors will be presented. The proposed method takes into account information which is usually ignored by timing verifiers. Thus, it is possible to identify correct signal flow directions and eliminate false paths for the problematic circuits during timing verification.

A switch level logic simulator [BaTr80, Brya84] is usually slower than a gate level logic simulator because the logic values of many more nodes in the circuits must be determined at the switch level. Therefore, if a switch level logic simulator evaluates only the logic values of those nodes which it is interested in and ignores the logic values of the

other nodes in the circuit, then the total simulation time can be reduced. This is because extra simulation time will not be spent for those nodes which are uninteresting. When performing logic simulation on the circuit represented in SLICE, only the logic values of interesting nodes are considered. Therefore, the proposed logic simulation method based on SLICE can result in a faster simulator.

1.4 Overview of the Dissertation

A digital MOS circuit is a network composed of transistors and nodes. Therefore, there is a natural way to express the transistor network structure in terms of a graph model. A graph representation of MOS circuits will be introduced in Chapter 2. A circuit representation, namely structured logical circuit expression (SLICE), is proposed to represent both structural and functional properties of MOS circuits. Once a SLICE of a circuit is available, the Boolean behavior of the circuit can be derived. The derivation of Boolean equations of a circuit expressed in SLICE representation will be presented.

Rules for extracting the Boolean equations of the different design styles of a circuit in SLICE will be presented in Chapter 3. To ensure that the circuit being extracted is electrically safe, an approach is introduced to verify that the underlying circuit is safe. The extraction of the general design style of static MOS circuits will also be examined.

Algorithms using the rules in Chapter 3 to extract the functionality of a transistor group are proposed in Chapter 4. First, the algorithm to extract the circuits which can be represented by directed acyclic graphs is exploited. Then the algorithm to extract the circuits represented by general directed graphs is developed. After the functionalities of all the subcircuits in a circuit are obtained, methods for determining the final aggregated functionality of the circuit will be presented.

The manner of performing timing verification for the circuits expressed in SLICE will be investigated in Chapter 5. A method to identify the correct signal flow direction

within a transistor group during timing verification are proposed. Furthermore, a heuristic to accelerate this process is presented. In Chapter 6, logic simulation at the switch level for the circuits in SLICE representation will be demonstrated.

A closing chapter will summarize the contribution of this research to the technique of MOS circuit verification. Finally, directions are suggested for future research in this area.

Chapter 2

MOS Circuit Representation

2.1 Introduction

In this chapter, the digital MOS circuit (or simply MOS circuit) model employed in this work is defined. An undirected graph representation to represent the MOS circuit interconnection will be discussed. A directed graph representation to describe the functional behavior of MOS circuits will also be presented. An expression, namely Structured LogIcal Circuit Expression (SLICE), to represent the connectivity and functionality of MOS circuits will be introduced based on the directed graph representation. A method to obtain the circuit functionality from a SLICE will be demonstrated after the introduction of SLICE. It is important to point out that SLICE may be used as a design representation and the designers themselves can specify the SLICE for the circuits being designed. However, it is also important to derive the SLICE from structural information such as circuit layout, and then compare the derived SLICE with the one specified by the designers to verify the correctness of the design.

2.2 MOS Circuit Model and Its Representation at the Switch Level

A MOS circuit $\Omega(N, T)$ is a *transistor network* consisting of a number of interconnected MOS transistors including pMOS transistors and nMOS transistors. T is the set of

MOS transistors in the circuit, and N is the set of nodes which interconnect MOS transistors. Each transistor has three terminals (nodes): *source*, *drain*, and *gate*. Source and drain terminals are called *data terminals*. In the switch level model of MOS circuits, a pMOS (nMOS) transistor is closed, or conducting, if the logic level of its gate terminal is 0 (1). When a transistor is conducting, the signal at one data terminal will pass through the transistor and reach the other data terminal. Thus, a *channel* is formed between two data terminals. On the other hand, a pMOS (nMOS) transistor is open, or not conducting, if the logic level of its gate terminal is 1 (0). In this case, the two corresponding data terminals are disconnected. The transistors in the switch level model act like bidirectional switches so that a signal may pass between their two data terminals in either directions.

Many properties in a MOS circuit can be revealed by identifying different types of nodes in the circuit. In a MOS circuit, we define the nodes which are important in the transistor network.

Definition 2.1: The following nodes in N of a MOS circuit $\Omega(N, T)$ are defined.

(1) An *external node* is V_{dd} (1), V_{ss} (0), or a node to which an external signal can be applied.

(2) All other nodes besides external nodes are *internal nodes*.

(3) A *gate node* is a node which is a gate terminal of one or more transistors.

(4) If a node is both an internal node and a gate node, it is called an *internal gate node*.

□

Note that an external node accepts any external signal applied to a transistor network. Note also that the "output" node referred in the network is never an external node.

When performing logic simulation, or functionality extraction, we want to know the states of each node in the circuit. In a MOS circuit, there are possible states in each node of the circuit. We define different states in the MOS circuit as follows:

Definition 2.2:

(1) *Logic 1* state (*logic 0* state) of a node in a MOS circuit is obtained from the signal coming from V_{dd} (1) (V_{ss} (0)), or external nodes which provide 1 (0), through a sequence of conducting transistors.

(2) If both states of logic 1 and logic 0 cannot occur at a node, then the state of this node is *high impedance*

(3) If both logic 1 state and logic 0 state can exist at the same node at the same time, then the state of this node is *unsafe*.

□

A node is called an *unsafe node* if an unsafe state can occur at this node. Otherwise it is a *safe node*. The four states defined in the above can handle static MOS circuit design (which is defined below) and can be easily extended to cover general static MOS circuits which allow a signal to be weakened by transistors in the circuit. We now formally introduce a class of MOS circuits, namely *static MOS circuits*. Static MOS circuits are the main circuits dealt with in this research. We have the following definition:

Definition 2.3: A MOS circuit is a *static MOS circuit* if it obeys the following two constraints:

1. Only four possible states are allowed in the nodes of the MOS circuit. The set of states is { 1, 0, Δ, U} which corresponds to logic 1, logic 0, high impedance, and unsafe, respectively.
2. It is not allowed in the MOS circuit that a high impedance state is applied to a gate node.

□

From the above definition, high impedance state can appear at data terminal nodes, but not at gate terminal nodes. For greater clarity, we will express the MOS circuit in terms of a mathematical structure. In the following subsection, we will use graph

structure to present our concepts.

2.2.1 Graph Representation of MOS Circuits

Our ideas can be more easily explained if the transistor network being discussed is viewed as an undirected graph. Actually, it is natural to think of a MOS circuit as an undirected graph. We have the following definition:

Definition 2.4: A MOS.circuit $\Omega(N, T)$ can be viewed as an undirected graph $G(V, E)$ called *channel graph*, where each vertex v in $V(G)$ has a one-to-one correspondence to a node in N, and each edge e in $E(G)$ a transistor in T [Brya84, RaTr87].

□

We can easily define different types of vertices in the channel graph from the corresponding MOS circuit.

Definition 2.5: In a channel graph $G(V, E)$, the following vertices in $V(G)$ are defined.

(1) A vertex is an *external vertex* if it is an external node in the MOS circuit.

(2) A vertex is an *internal vertex* if it is an internal node in the MOS circuit.

(3) A vertex is a *gate vertex* if it is a gate node in the MOS circuit.

(4) A vertex is an *internal gate vertex* if it is an internal gate node in the MOS circuit.

□

Note, for the sake of convenience, when referring to a graph, vertex and node will be used interchangeably. For the circuit in Figure 2.1 (a), nodes *a*, *b*, 0, and 1 are external nodes. Nodes *a*, *z* and *b* are gate nodes. Node *z*, *g*, and *f* are internal nodes. Nodes *z* is an internal gate node.

It is quite natural to partition a MOS circuit into a number of subcircuits so that in each of subcircuits, all the transistors are channel-connected through internal nodes

[Brya84, RaTr87]. For the transistor network partitioning purpose, it must be ensured that external vertices, if they are data terminal nodes of some transistors in the circuit, have vertex degree 1. Therefore, these external vertices must be replicated in the channel graph as needed. We define the induced channel graph as the graph obtained after partioning the channel graph.

Definition 2.6: Given a channel graph, a new graph can be obtained if we replicate all the external vertices, that are data terminal nodes of transistors, such that each external vertex has vertex degree 1. The resulting graph after replication is the *induced channel graph* of the channel graph [Brya84, RaTr87].

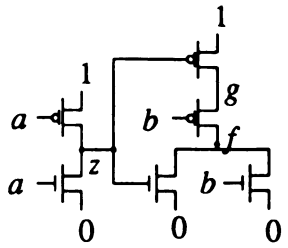
□

By replicating the external nodes which are data terminal nodes of transistors in the circuit, we have already performed a partition on the MOS circuit. The MOS circuit is then partitioned into subcircuits as defined in the following:

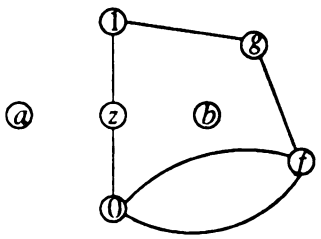
Definition 2.7: A MOS circuit can be partitioned into a number of *transistor groups*, where each transistor group is a *connected component* [AhHU83] of its corresponding induced channel graph. If a connected component in an induced channel graph contains only a vertex, then it is a *trivial* connected component, otherwise, it is a *non-trivial* connected component.

□

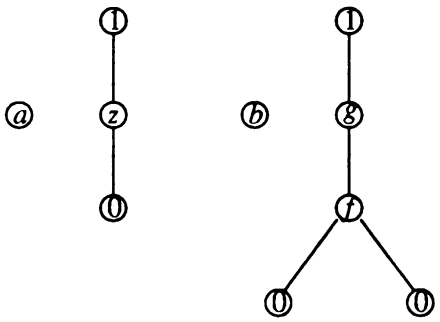
Figure 2.1 (a) shows a simple MOS circuit, and its corresponding graph representation is shown in Figure 2.1 (b). Obviously, as shown in Figure 2.1 (c), the induced channel graph has four isolated subgraphs after its external vertices are properly replicated. We will introduce the direction concept for transistors in the circuit. It is natural to use directed graph representation to describe this concept. In the following subsection, it shows how we can go from an undirected graph to a directed graph to represent a MOS circuit.



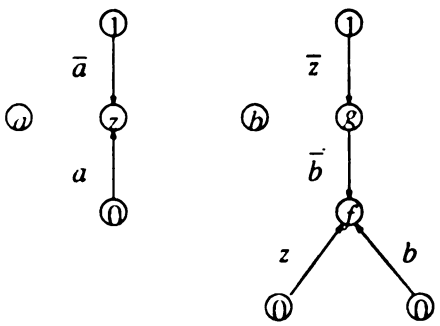
(a) MOS circuit



(b) Corresponding channel graph representation



(c) Four connected components in induced channel graph



(d) LDCCs w.r.t. nodes f and z

Figure 2.1 Graph representation of a MOS circuit

2.2.2 Directed Graph for MOS Circuits

In this section, a directed graph representation is presented to describe the behavior of MOS circuits. It is necessary to introduce some basics from graph theory before we proceed. A directed graph with no cycles is called a *directed acyclic graph* (DAG). A *directed path* in a directed graph is a sequence of edges $e_1, \cdots, e_k$ where each edge e_i is of the form (u_i, u_{i+1}), u_i and u_{i+1} are vertices in the graph and $1 \leq i \leq k$ such that every vertex in the path is distinct. k is the *length* of the directed path. u_1 and u_{k+1} are called the *origin* and *terminus* of the path, respectively, while the vertices $u_2, u_3, \cdots, u_k$ are its *internal vertices*. A directed path is a *directed cycle* if $u_1 = u_{k+1}$ and $k > 1$.

A *labeled directed graph* is a directed graph with a label for each edge in the graph. An *l-path* corresponding to a directed path $p = < e_1, \cdots, e_k >$ in a directed graph is a sequence of labels $< l_1, \cdots, l_k >$ where l_i is the label of edge e_i for all $1 \leq i \leq k$. The *length* of an l-path is the length of the corresponding directed path. A *product term* of an l-path $<l_1, \cdots, l_k>$ is $l_1 \cdots l_k$. An l-path or product term is said to *disappear* if its corresponding product term can become Boolean 0 after applying Boolean operations on it by considering each label as a Boolean variable and a product term as a Boolean product of Boolean variables.

For any node but external nodes in a transistor group, it is desirable to identify the functionality of this node, especially an output node. To determine the functionality of a particular node in a transistor group, namely the *goal node*, it is helpful to identify all the signal paths from the external nodes toward the goal node through a sequence of transistors. Though MOS transistors are physically bidirectional, we may define the direction of each transistor in the circuit in terms of their signal path directions. This direction of MOS transistors allows transistors to be unidirectional, bidirectional, or non-directional.

Definition 2.8:

(1) The *direction* of each transistor in a transistor group with respect to a goal node is defined by all the path directions from the external nodes through a sequence of transistors toward this goal node.

(2) After all the possible signal path directions are considered with respect to the goal node, the direction of a transistor in a transistor group may become *unidirectional* if all the path directions are from one data terminal to the other data terminal of the transistor, *bidirectional* if the paths through the transistor are from either one of the data terminals to the other, or *non-directional* if no path goes through this transistor.

□

Obviously, the direction of a transistor is dependent on the location of the goal node, the node whose functionality is to be extracted. Thus, no external node can be a goal node. Usually the goal node is a gate node or an output node to be observed. If it is a gate node, it is normally the gate terminal of transistors in other transistor groups.

We have defined the direction of each transistor in a transistor group with respect to a goal node. Thus, we can now define a directed graph for each connected component in an induced channel graph.

Definition 2.9: A *directed connected component* (DCC) with respect to a goal node is a directed graph which is the nontrivial connected component of an induced channel graph with directed edges. The direction of each edge with respect to goal node in the DCC is the same as the corresponding transistor direction in the transistor group with respect to the same goal node. If an edge in the connected component corresponds to a bidirectional transistor, a new edge will be added in parallel to the original edge and these two edges are of different directions.

□

The gate terminal signal of each transistor in the circuit controls the on and off of the transistor. Thus, signal names of gate nodes are important. We now have the following definition:

Definition 2.10: *Labeled DCC* (LDCC) of a transistor group is a DCC with labeled edges, where the labels are the signal names of the gate nodes for n-type transistors corresponding to these edges, or the negation of the signal names for p-type transistors.

□

Note that the signal name of the gate node of a transistor may be either an external input signal name (input variable name) or an internal gate node name. Two special external signal names are 1 (V_{dd}) and 0 (V_{ss}). In the example of Figure 2.1, to find out the functionality of node *f*, the Boolean behavior of node *z* must be known. Thus, *z* is the goal node in its group. All the transistor directions are determined by the signal path traversed, starting from external nodes to node *z*. Similarly, the transistor directions with respect to node *f*, which is a goal node in its group, can be determined. Figure 2.1 (d) is a directed graph with labels. It has two LDCCs and shows the directions of transistors with respect to the goal nodes *f* and *z*, respectively. In the corresponding induced channel graph of a transistor network, the external nodes all have node degree 1 or 0. The goal nodes and gate nodes in channel graph *G* can be identified from the original transistor network Ω.

To determine the direction of the transistors with respect to a given goal node, all the signal paths from the external nodes toward the goal node must be found. A pathfinding algorithm based on the depth first search can be utilized to search for these paths. The search is done until all the signal paths are found. The direction of each transistor is decided by the direction of the path from the external nodes to the goal node.

Particularly interesting are special nodes called *junction nodes*. A junction node is the node which V_{dd} and V_{ss}, V_{dd} and an input signal, V_{ss} and an input signal, or two input

signals, may reach by travelling along two different signal paths. Since the signals of different logic states may reach junction nodes, the behavior description of junctions nodes is of importance. Actually, junction nodes play a key role in functionality extraction as shown in the following chapters. Junction nodes can be formally defined as follows:

Definition 2.11: Given an LDCC, let two directed paths in this LDCC be $p_1 = < e_1, \cdots, e_k >$ and $p_2 = < f_1, \cdots, f_l >$ where $e_i = (u_i, v_i)$, $1 \le i \le k$, $f_j = (w_j, x_j)$, $1 \le j \le l$, $e_i \neq f_j$, and $v_k = x_l$. Vertex v_k is a *junction vertex* if u_1 and w_1 are both external vertices which have different signal names. The corresponding nodes in the transistor network of junction vertices are called *junction nodes*. If all the internal vertices in p_1 and p_2 are not junction vertices, then p_1 and p_2 are called the *prime paths* of junction vertex v_k. u_1 and w_1 are the origins of the prime paths.

□

Note that in the above definition, vertex v_k is also a junction vertex if u_1 is either an external vertex or a junction vertex, which has a different signal name from w_1, where w_1 is either an external vertex or a junction vertex. The following example shows the junction nodes in a given LDCC.

Example 2.1: An LDCC is shown w.r.t. node f as shown in Figure 2.2. 1, 0, a, b, and c are external input signals. The nodes n_5, n_7, n_{10} and n_{12} are junction nodes by the definition, since different logic states may reach these nodes. The prime paths are $<(n_1,n_3),(n_3,n_5)>$, $<(n_2,n_3),(n_3,n_5)>$, $<(n_4,n_5)>$, $<(n_5,n_7)>$, $<(n_6,n_7)>$, $<(n_7,n_8),(n_8,n_{12})>$, $<(n_9,n_{10}>$, $<(n_{11},n_{10})>$, and $<(n_{10},n_{12})>$.

□

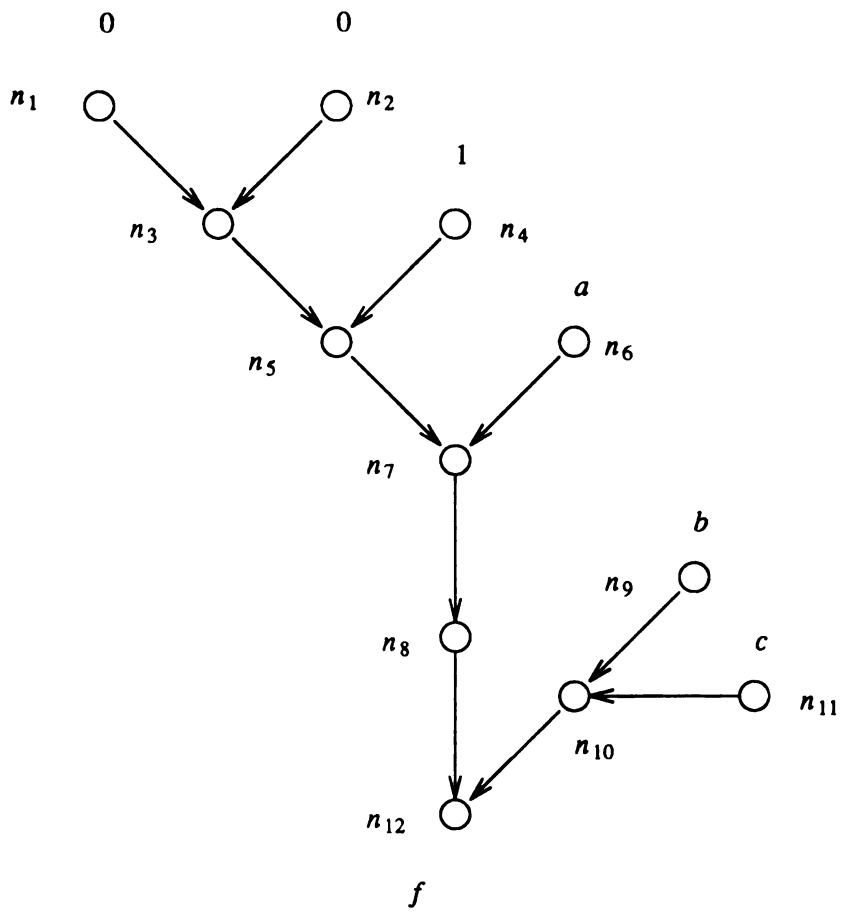


Figure 2.2 LDCC w.r.t. node *f*

2.3 SLICE Representation and Boolean Equation of MOS Circuits

Functional information from the circuit allows us to understand the behavior of the circuit. Thus, if we could have a circuit representation which carries not only the structural but also functional information of the circuit, then we would have a better understanding of the circuit behavior and an accurate knowledge of the functionality of the circuit. This kind of representation can be used as a circuit design representation since it describes both circuit connectivity and functionality. When transistor direction information is available in a MOS circuit, functional information can be obtained from the circuit. Since designers have the transistor direction of the circuit in mind, they can add this knowledge to a design representation if there is a representation that can accommodate

this information. In this section, a circuit representation is proposed which can provide circuit information for both connectivity and functionality. The derivation of the functionality of the circuit from this representation is presented.

2.3.1 SLICE Representation

Once all the transistor directions of a group have been decided with respect to a goal node, all non-external nodes can be described in terms of their neighboring nodes and neighboring transistors. Thus, a Structured LogIcal Circuit Expression (SLICE), which expresses both circuit connectivity and signal flow direction, is proposed to represent each internal node in the circuit.

Definition 2.12: Given the LDCC of a transistor group with respect to a goal node, the *Structured LogIcal Circuit Expression* (SLICE) of an internal node z in this LDCC is defined as

$$E_z = \sum_{j=1}^{m} \left(\left(\prod_{i=1}^{n_j} g_{ji} \right) d_j \right) \tag{2.1}$$

where $\prod_{i=1}^{n_j} g_{ji}$ is the product term of $<g_{j1}, \cdots, g_{jn_j}>$, the l-path of a directed path starting from node j and ending at node z. Nodes j are origins of the directed path. n_j is the length of the l-path. d_j is the signal name of node j, and m is the in-degree of node z.

□

The SLICE of a node z is *self-defined* if one g_{ji} in (2.1) is the signal name or the negation of the signal name of node z. We will pay special attention to those circuits with self-defined SLICE in Chapter 3. Furthermore, for simplicity of notation, z is used instead of E_z for a node z's SLICE in (2.1) as shown below. Also, we may use signal name d_j in (2.1) to refer to a node j for the sake of convenience.

Since each edge of a LDCC corresponds to a transistor, attributes of each transistor may be associated with each edge in the LDCC, and thus with each label in the SLICE. Those attributes which may be useful for verification purposes include a transistor's type, size, gate terminal node label, or identifying name. Similarly, the attributes related to each node in the circuit may be associated with the vertex in the LDCC. Those attributes may include the state of the node, node types as defined in Definition 2.1, or the capacitance of the node.

Note that this definition of SLICE is from an LDCC which has signal flow direction information. However, this direction information may be specified by the designer who designs the circuit. Thus, the definition of SLICE can also be used for the circuit in which signal flow information is already known. From this point of view, SLICE can be used as a design representation to represent a design in which both structural and functional information are embedded. From the definition, we know that the SLICE of a transistor group depends on the location of the goal nodes as shown in the following example.

Example 2.2: According to (2.1), for the circuit shown in Figure 2.3 (a), we have SLICEs for nodes f and g, respectively.

From the LDCC with respect to node f shown in Figure 2.3 (b), we have the SLICE of node f

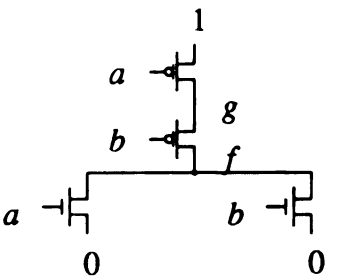
$$f = a0 + b0 + \bar{b}\bar{a}1$$

From the LDCC with respect to node g shown in Figure 2.3 (c), we have the SLICE of node g

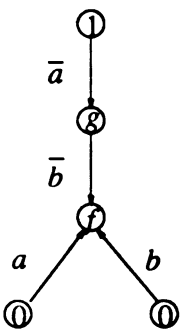
$$g = \bar{b}a0 + \bar{b}b0 + \bar{a}1 = \bar{b}a0 + \bar{a}1$$

Note that f and g are used instead of E_f and E_g.

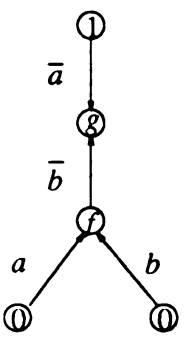
□



(a) MOS circuit



(b) LDCC w.r.t. node f



(c) LDCC w.r.t. node g

Figure 2.3 MOS circuit and its LDCCs

Once we have the SLICE of a transistor group, we can derive from it the functional behavior of the group. The next subsection describes how to obtain the corresponding Boolean equations from a given SLICE.

2.3.2 Boolean Equation of MOS Circuits

Given the SLICE expression of an internal node, Boolean equations of this node can be obtained according to its SLICE. These Boolean equations characterize the functionality of this node. Since high impedance state is allowed in MOS circuits, it is necessary to have three Boolean equations instead of only one to characterize the functionality of the node under discussion. For the node d_j in a given SLICE of a MOS circuit, three Boolean equations $(d_j)_1$, $(d_j)_0$, and $(d_j)_\Delta$ are defined at this node for Boolean logic 1, Boolean logic 0, and Boolean logic Δ, respectively. The value of Boolean equation $(d_j)_1$ becomes logic 1 only when logic 1 state occurs at this node. Its value is logic 0 when no logic 1 state occurs at this node. Similarly, Boolean equation $(d_j)_0$ and Boolean equation $(d_j)_\Delta$ are defined for logic 0 state and high impedance state, respectively.

Definition 2.13: Given a SLICE of a node z in (2.1), the three corresponding Boolean equations are defined as

$$z_1 = \sum_{j=1}^{m} \left(\left(\prod_{i=1}^{n_j} g_{ji} \right) (d_j)_1 \right) \tag{2.2}$$

$$z_0 = \sum_{j=1}^{m} \left(\left(\prod_{i=1}^{n_j} g_{ji} \right) (d_j)_0 \right) \tag{2.3}$$

$$z_\Delta = \prod_{j=1}^{m} \left(\sum_{i=1}^{n_j} \overline{g_{ji}} + (d_j)_\Delta \right) \tag{2.4}$$

□

Boolean logic 1 equation z_1 is defined by all possible signal paths starting from logic 1 state of each node d_j, and Boolean logic 0 equation z_0 is defined by those paths starting from logic 0 state of each node d_j. Boolean logic Δ equation z_Δ is used to indicate that a node is in high impedance state. Whenever there is no conducting path from external nodes to a particular node, then this node is in high impedance state. Note that in the following discussion, it is assumed that all external inputs are either in logic 1 state

or in logic 0 state, but not allowed in high impedance state. The following examples show how to derive the corresponding Boolean equations when a SLICE is given.

Example 2.3: We want to determine Boolean equations of nodes f and g in the Example 2.2. Since $(1)_1 = 1$, $(1)_0 = 0$, $(0)_1 = 0$, and $(0)_0 = 1$, and it is also assumed, as mentioned before, that no high impedance state can occur at external nodes 0 and 1, i.e. $(1)_\Delta = 0$, $(0)_\Delta = 0$. thus from (2.2)-(2.4), we have

$$f_1 = \bar{b}\,\bar{a} \quad f_0 = a + b \quad f_\Delta = (a + b)\,\bar{a}\,\bar{b} = 0$$

$$g_1 = \bar{a} \quad g_0 = \bar{b}a \quad g_\Delta = a\,(b + \bar{a}) = ab.$$

Note that the SLICE of node g is based on the transistor direction with respect to g. Thus, node g may be in high impedance state if a is in logic 0, and b logic 1.

□

Example 2.4: Find the Boolean equations of node f in Figure 2.4 if Boolean equations of both nodes m and n are given. If m_1, m_0, m_Δ, n_1, n_0, and n_Δ are known, then from (2.2)-(2.4), we have

$$f_1 = (ab)\,m_1 + (cd)\,n_1$$

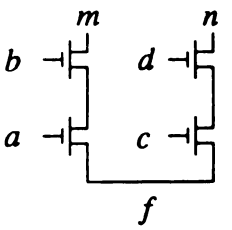
$$f_0 = (ab)\,m_0 + (cd)\,n_0$$

$$f_\Delta = (\bar{a} + \bar{b} + m_\Delta)(\bar{c} + \bar{d} + n_\Delta).$$

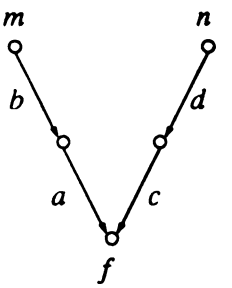
□

Similarly, if more signals reach node f in the above example, similar formulae can be derived. For static MOS circuits with self-defined SLICEs, the derivation of the corresponding Boolean equations may not be straightforward as shown in (2.2)-(2.4). However, it is still possible to derive the Boolean equations for them. This will be demonstrated in the next chapter.

We have explained how to derive Boolean equations from a SLICE. However, it still needs further investigation to see if the derived equations can indeed correctly



(a) MOS circuit



(b) LDCC w.r.t. node *f*

Figure 2.4 MOS circuit and its LDCC

represent the circuit. In the following Chapter 3, rules are proposed to examine the correctness of the Boolean equations derived from a SLICE. These rules also assist in deriving the Boolean equations of different circuit design styles.

Chapter 3

Extraction Rules

3.1 Introduction

Formal verification through Boolean comparison is a useful technique to verify the functional correctness of a circuit. This method compares the Boolean behavior extracted from the circuit layout with the behavior from the design specification to determine whether both are equivalent. However, because of the bidirectionality and high impedance state of MOS transistors, the Boolean behavior extraction method used in traditional logic gate design cannot be adequately applied to VLSI MOS circuits.

An efficient functionality extraction method for static MOS circuit and some dynamic circuits based on SLICE representation will be presented. Several verification rules are proposed to extract the Boolean behavior of MOS transistor circuits through a fast guiding algorithm. Section 2 will present the condition to verify whether the nodes in the circuit are electrically safe. The rules to do the extraction are demonstrated in Section 3, 4, and 5, while the extraction algorithm is presented in the next chapter. In this proposed approach, a transistor circuit is first partitioned into a number of transistor groups, where each transistor group is an isolated connected component of the corresponding induced channel graph. The Boolean behavior of each group is then extracted. The whole circuit behavior can be obtained by aggregating the behavior of all

the individual groups. The whole verification scheme is shown in Figure 3.1. This work deals with the first three processes of Figure 3.1, circuit partitioning, functionality extraction, and functionality aggregation. The last functionality comparison process is not considered here and can be found in [OTOO86, WeSa86].

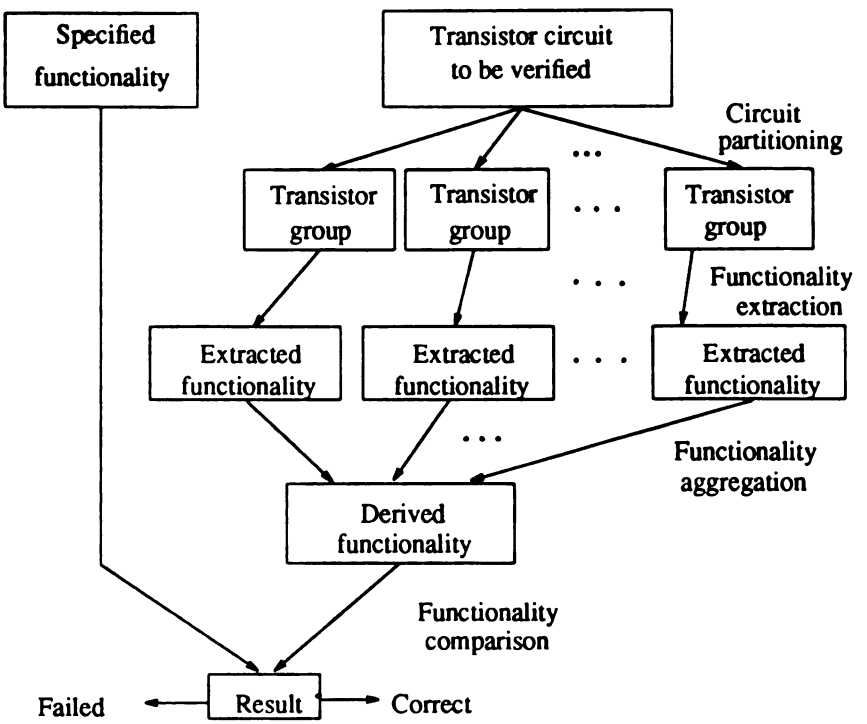


Figure 3.1 A functional verification scheme

3.2 Electrical Property Consideration

To extract the functionality of a circuit, it is desirable to guarantee that there is not only no functional error but also no electrical error (i.e., no unsafe nodes). Thus, some electrical rule checking is needed to uncover electrical errors. First we have the following lemma.

Lemma 3.1: Given the SLICE of a node f, and the corresponding Boolean equations f_1 and f_0. If $f_1 f_0 = 0$, then this node is safe.

□

Proof: Since $f_1 f_0 = 0$, logic 1 and logic 0 cannot reach node f at the same time. Thus, node f is safe.

□

The following theorem can help examine the safe nodes in a transistor group.

Theorem 3.1: Given the SLICE of a node f, $f = f_0 0 + f_1 1$, if all the product terms do not disappear and $\overline{f_0 + f_\Delta} = f_1$, where f_0, f_1, and f_Δ are Boolean logic 1, Boolean logic 0, and Boolean logic Δ equations of node f's SLICE, respectively, then all the nodes present in f's SLICE are safe.

□

Proof: Because $\overline{f_0 + f_\Delta} = f_1$ at node f, logic 1 and logic 0 will not reach node f. This implies $f_1 f_0 = 0$. Hence, node f is safe. Suppose that there exists an unsafe node, say node x, present in f's SLICE. Since this node is unsafe, both logic 1 and logic 0 may reach this x node at some moment. Because no product term disappears, there is a path from node x to node f. Thus, the unsafe state of node x may reach node f through this path. Hence node f is unsafe, and a contradiction.

□

This theorem shows that if the safe condition is satisfied for a portion of a circuit, then this partial circuit is safe. However, if the condition is not satisfied, then this partial circuit may or may not be safe.

Example 3.1 We want to check if the circuit in Figure 3.2 is safe. The SLICE of node c is

$$c = \bar{b}\ 1 + b\ a\ 1 + b\ \bar{a}\ 0$$

Since, $\overline{c_0} = c_1$ and no product term disappears, thus the whole circuit is safe.

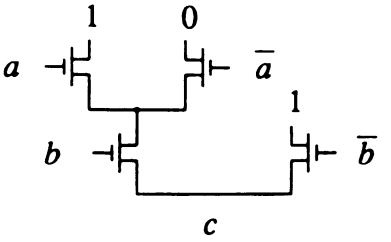


Figure 3.2 Safe circuit

□

Example 3.2: Is the circuit in Figure 3.3 safe?

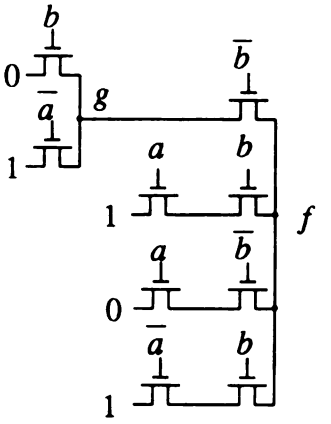


Figure 3.3 Unsafe circuit

The SLICE of node f is

$$f = b\bar{a}1 + \bar{b}a0 + ba1 + \bar{b}b0 + \bar{b}\bar{a}1$$

$$= b\bar{a}1 + \bar{b}a0 + ba1 + \bar{b}\bar{a}1$$

$$= (b\bar{a} + ba + \bar{b}\bar{a})1 + (\bar{b}a)0$$

$$= f_1 1 + f_0 0$$

Although $\overline{f_0} = f_1$, one product term disappears in f. Thus, the safe condition is not satisfied, and the nodes in the circuit cannot be guaranteed to be safe. Actually, an electrical short may occur at node g.

□

Example 3.3: The circuit in Figure 3.4 has two groups. The left group is safe because $b = \bar{a}\ 1 + a\ 0$ and $\overline{b_0} = b_1$. For the other group $f = a\ 1 + b\ 0$, if we do not know that $b = \bar{a}$, we may conclude that f is unsafe, since $\overline{f_0} \neq f_1$. However, since $b = \bar{a}$, after substituting b by $\bar{a}$, we know that f is safe.

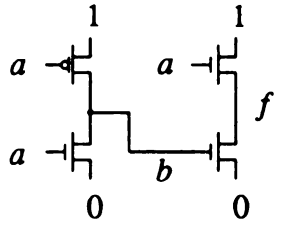


Figure 3.4 Circuit with two transistor groups

From this example, we know that if a group is determined to be safe using internal signal names, then it is safe. Otherwise, it is necessary to substitute external signal names for all internal signal names. After substitution if it is still unsafe, then we can draw the conclusion that this node is really unsafe.

□

In general, if the internal signal names appear in both of the Boolean equations being compared, then it is not necessary to do the substitution. However, if only one equation has some signal names which do not appear in the other equation, then we have to substitute internal signal names until both equations have all the same signal names.

3.3 Rules for Functionality Extraction

Several rules will be proposed to extract different design styles of static MOS circuits in Section 3.3.1. The way to extract the circuit with a self-defined SLICE will be explained in Section 3.3.2. In Section 3.3.3, a general extraction rule will be demonstrated for general static MOS circuits.

3.3.1 Extraction Rules

In some MOS design styles, such as pass transistor logic and transmission gate design, input signals may be applied to data terminals. Therefore, d_j in (2.1) may not be 0 or 1. Some method is needed to transform the SLICE with input signals into another form which will aid the Boolean equations extraction. The following rule achieves the transformation.

Rule 1: If the SLICE of a node z is $(\prod_i x)\, a$ and a is either logic 1 or logic 0, then the Boolean equations of node z is $z_0 = (\prod_i x)\,\bar{a}$, $z_1 = (\prod_i x)\, a$, $z_\Delta = \sum_i \bar{x}$.

□

Justification: For the node a, we have $a_1 = a$, $a_0 = \bar{a}$ and $a_\Delta = 0$. Thus, the above Boolean equations can be derived from formulas (2.2)-(2.4).

□

Example 3.4: Consider in Figure 3.5 the pass transistor logic with two transistors, where a is an input signal with either logic 1 or logic 0, and f is the output to be observed.

The SLICE of node f is $f = \bar{c}ba$. Thus, from Rule 1, we have $f_0 = \bar{c}b\bar{a}$, $f_1 = \bar{c}ba$, and $f_\Delta = c + \bar{b}$.

□

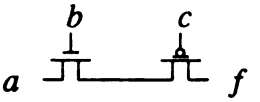


Figure 3.5 Pass transistor logic design

From Theorem 3.1 in the previous section, the following rule is used to check whether a node is safe or not, based on the node's SLICE. Note that it is always assumed that the external input is either logic 1 or logic 0, but not high impedance state.

Rule 2: Given the SLICE of node f, $f = f_0 0 + f_1 1$. If $\overline{f_0 + f_\Delta} = f_1$, then the node f is safe; otherwise, the node f is unsafe. Furthermore, if no product term disappears in f's SLICE, then all the nodes present in the SLICE of f are safe. Also, if $f_\Delta = 0$, then one Boolean equation f_1 itself is sufficient to represent the functionality of the node f.

□

Justification: Justified from Theorem 3.1.

□

Note that if $f_0 = 0$, then f_1 and f_Δ are used to represent the functionality. Similarly, if $f_1 = 0$, the functionality is represented by f_0 and f_Δ. Rule 2 provides a step by step way to examine the unsafe nodes in a transistor group. If it is established that the nodes in a portion of the circuits are safe, then we only need to examine the nodes in the rest of the circuit. In fact, a lot of fully complementary CMOS structures, such as logic gates, can be examined by Rule 2, and most of them turn out to be safe without logic Δ state.

Example 3.5: In Figure 3.6, the circuit is a pass transistor logic, the node f is an exclusive or function, XOR, where a and b are input signals.

According to Rule 1, we have

$$f = \bar{b}a + b\bar{a} + \bar{a}b + \bar{a}b$$

$$= \bar{b}a + b\bar{a} + \bar{a}b$$

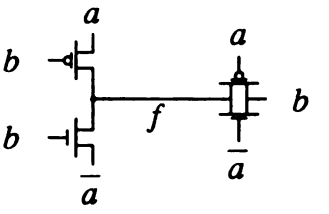


Figure 3.6 XOR circuit

$$= (\bar{b}\bar{a}0 + \bar{b}a1) + (ba0 + b\bar{a}1) + (\bar{a}\bar{b}0 + \bar{a}b1)$$

$$= (\bar{b}\bar{a} + ba)0 + (\bar{b}a + b\bar{a})1$$

Thus, $f_0 = \bar{b}\bar{a} + ba$, $f_1 = \bar{b}a + b\bar{a}$, and $f_\Delta = (b + a)(\bar{b} + \bar{a})(b + \bar{a})(\bar{b} + a) = 0$. Since $\overline{f_0} = f_1$ and no product term disappears, from Rule 2 the circuit is safe and f_1 itself is sufficient to represent the functionality of the XOR circuit.

□

Also, it can be shown that the circuit in Example 2.2 is safe because $\overline{f_0} = f_1$, and no product term disappears. Actually, f_1 can represent the functionality of a two-input NAND gate. By applying Rule 2 in a divide-and-conquer fashion, we can determine whether the whole circuit is safe or not, as demonstrated in the following example.

Example 3.6: In the circuit of Figure 3.7, we want to examine the functionality of node g, and whether the whole circuit is safe or not.

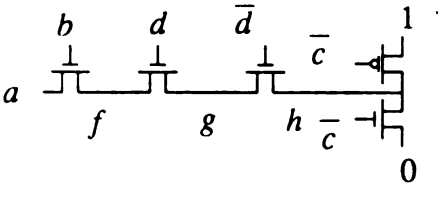


Figure 3.7 Circuit with logic Δ state

We may consider all the paths reaching g. Thus, we have

$$g = dba + \bar{d}c1 + \bar{d}\bar{c}0$$

$$= dba1 + db\bar{a}0 + \bar{d}c1 + \bar{d}\bar{c}0$$

$$g_1 = dba + \bar{d}c$$

$$g_0 = db\bar{a} + \bar{d}\bar{c}$$

$$g_\Delta = (\bar{d} + \bar{b} + \bar{a})(\bar{d} + \bar{b} + a)(d + \bar{c})(d + c) = d\bar{b}$$

Because $\overline{g_0 + g_\Delta} = g_1$, node g is safe. From Rule 2 we can draw the conclusion that the whole circuit is safe. However, if we do go through intermediate nodes f and g, then we can also answer the question of whether the whole circuit is safe or not. Consider the SLICEs of nodes f and h first.

$f_1 = ba \quad f_0 = b\,\bar{a} \quad f_\Delta = \bar{b}$. Since $\overline{f_0 + f_\Delta} = f_1$, node f is safe.

$h_1 = c \quad h_0 = \bar{c} \quad h_\Delta = 0$. Since $\overline{h_0 + h_\Delta} = h_1$, node h is safe.

$$g_1 = df_1 + \bar{d}h_1 = dba + \bar{d}c$$

$$g_0 = df_0 + \bar{d}h_0 = db\bar{a} + \bar{d}\bar{c}$$

$$g_\Delta = (\bar{d} + f_\Delta)(d + h_\Delta) = df_\Delta + \bar{d}h_\Delta + f_\Delta h_\Delta = d\,\bar{b}$$

Since $\overline{g_0 + g_\Delta} = g_1$, node g is safe. Thus, the whole circuit is safe and the behavior of node g is described by g_0, g_1, and g_Δ.

□

In VLSI circuit design, a design style called *pseudo-nMOS* logic is used in many CMOS PLA designs. It uses pull-up transistors to introduce a weakened signal from V_{dd}. A pull-up node is never trapped into a high impedance state. It is considered to have logic 1 unless grounded through other paths. Thus, pseudo-nMOS circuits do not have high impedance states as found in other MOS logic structures. In general, if it is allowed that some attenuated signal can be overridden by the signal coming from other signal

sources through gate controlled conducting transistors, then the following rule is required.

Rule 3: If the SLICE of a node f is $f = f_0\ 0 + 1$ then the only Boolean equation of node f is $\overline{f_0}$. If $f = 0 + f_1\ 1$ then the only Boolean equation of node f is f_1. In both cases, $\overline{f_0}$ and f_1 are the traditional two-valued Boolean equations of node f.

□

Justification: Suppose $f = f_0\ 0 + 1$. Node f is always logic 1 if there is no conducting path from logic 0 (V_{ss}). Therefore, there is no high impedance state. Hence, we do not need a logic Δ equation, and only one logic 1 equation is sufficient to represent the functionality of node f. Since f_0 considers all the paths to get logic 0, $\overline{f_0}$ will consider all the paths to get logic 1. Thus, the Boolean equation of node f is $\overline{f_0}$. The second statement of Rule 3 can be similarly justified.

□

Note that before applying Rule 3, we must check that the overridden signal has been attenuated through some transistors. For instance, in the nMOS case, it uses a depletion pull-up transistor to decay the signal. Since in SLICE representation, transistor attributes, such as size, can be associated with each transistor, the check can be done before Rule 3 is applied.

Example 3.7: Consider in Figure 3.8 the pseudo-nMOS circuit with a bridging transistor.

Since the p-type transistor is always on, $f = (\ ad + be + ace + bcd\ )0 + 1$. Thus, the Boolean equation of node f is $\overline{ad + be + ace + bcd}$.

□

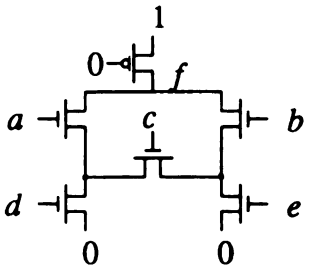


Figure 3.8 Pseudo-nMOS circuit

3.3.2 Extraction of Self-defined SLICE

We have not yet considered how to extract the functionality of a circuit whose SLICE is self-defined. If it is a self-defined SLICE, first, it must be established that the circuit behavior follows the constraints of a *static MOS circuit* (SMC) , and then the circuit behavior can be extracted by the following given theorem. We have the following lemmas.

Lemma 3.2: If the SLICE of a node z in (2.1) is self-defined, and for every j, $j = 1, \cdots, m$, at least one g_{ji} is the signal name or negation of the signal name of node z, then the circuit expressed by this SLICE is not an SMC.

□

Proof: Since at least one gate control of each signal path toward node z itself is controlled by node z, the logic value of node z may initially rely on the capacitance of node z instead of any signal from external inputs. Thus, since the circuit violates the first constraint of an SMC in Definition 2.3, the circuit is not an SMC.

□

Lemma 3.2 tells if a node's output feeds back to some gate controls of all the signal paths coming toward this node, then the circuit associated with this node is not an SMC.

Example 3.8: Given the circuit in Figure 3.9, the corresponding SLICE for node z is $E_z = za + zdb + ezc$. From Lemma 3.1, the circuit is not an SMC.

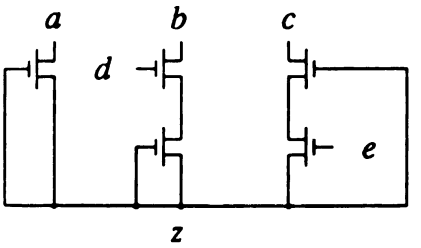


Figure 3.9 Circuit which is not an SMC

□

Lemma 3.3: If E_z, the SLICE of a node z in (2.1), is self-defined, then z_Δ must be 0.

□

Proof: Suppose $z_\Delta \neq 0$, then high impedance state would apply to the gate terminal node. This violates the second constraint of an SMC in Definition 2.3. Therefore, z_Δ must be 0.

□

We now present the following theorem. This theorem derives the Boolean equations of a circuit that has a self-defined SLICE.

Theorem 3.2: If E_z in (2.1) is self-defined, the following procedure can be used to determine whether the underlying circuit is an SMC and to compute the Boolean equations of node z if it is.

1. Let S_z be the SLICE of node z such that

$$S_z = \sum_{j=1}^{m} \left(\left(\prod_{i=1}^{n_j} g_{ji} \right) d_j \right) \quad \text{such that} \quad j \neq k ,$$

g_{ki} is the signal name of node z, $1 \leq i \leq n_k$,

2. If S_z is null then the circuit is not an SMC. Otherwise, compute z_1, z_0, and z_Δ from S_z.
3. If the z_Δ from S_z is not 0, then the circuit cannot be an SMC. If it is 0 then it is an SMC; thus substitute g_{ki}, which is the signal name of node z, in E_z by z_1, and recompute z_1 and z_0 from E_z. z_1 and z_0 are the Boolean expressions for node z.

□

Proof: In step 1, the product terms in E_z that cause a self-defined circuit are eliminated from E_z, thus the new SLICE is S_z. If S_z is null, then from Lemma 3.2, we know that the underlying circuit is not an SMC. Otherwise, z_1, z_0, and z_Δ can be derived from S_z. If z_Δ from S_z is not 0, then from Lemma 3.3, the circuit represented by S_z cannot be an SMC, and thus, the underlying circuit represented by E_z is not an SMC either. However, if z_Δ is 0, then we know that the underlying circuit is an SMC from Lemma 3.3. Therefore, from S_z, z_1 and z_0 can then be derived. Since z_Δ is 0, z_1 itself is the Boolean behavior which contributes to node z through other signal paths except those paths which cause the transistor group to be self-defined. Thus, after substituting g_{ki} in E_z with z_1 obtained from S_z, we can derive z_1 and z_0 from this new E_z, z_1 and z_0 are the Boolean expressions for node z.

□

Example 3.9: The behavior of node f in the circuit of Figure 3.10 is an XOR, and the Boolean expressions of node g are to be derived.

The SLICE of node g which is self-defined is

$$E_g = \bar{a}1 + a0 + b\bar{b}a + b\bar{a}b + bgb$$

$$= \bar{a}1 + a0 + b\bar{a}b + bgb$$

Taking out bgb, we have

$$S_g = \bar{a}1 + a0 + b\bar{a}b$$

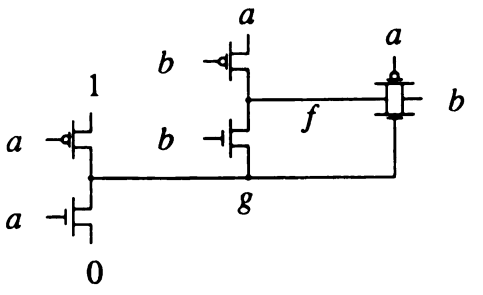


Figure 3.10 Circuit with self-defined SLICE

$$= \bar{a}1 + a0 + b\bar{a}\bar{b}0 + b\bar{a}b1 \quad \text{(By Rule 1)}$$

$$= \bar{a}1 + a0$$

From S_g, we have $g_1 = \bar{a}$, $g_0 = a$, $g_\Delta = 0$, thus this circuit is an SMC and $g = g_1 = \bar{a}$. Substitute g by g_1 in E_g.

$$E_g = \bar{a}1 + a0 + b\bar{a}b + b\bar{a}b$$

$$E_g = \bar{a}1 + a0$$

From Rule 2, we know node g is safe and we have $g_1 = \bar{a}$, and $g_0 = a$ which are the Boolean expressions for node z.

□

3.4 Generalized Extraction Rule 2

General static MOS circuits allow different strengths of signal paths in the circuits. The external nodes have the signal with the largest strength. The signals from the external nodes may be weakened through conducting transistors, so different strengths of signal may appear in a circuit. To extract the Boolean behavior from a circuit having different signal strengths. Theorem 3.1 and Rule 2 need to be generalized to handle the general static MOS circuits as shown in the following.

Suppose strength 1 is the strongest signal, and strength n the weakest signal where n is an integer greater than or equal to 1. We say a node is *safe* in general static MOS circuits if logic state 1 and logic state 0 of each signal strength i, $i = 1, \ldots, n$, cannot reach this node at the same time. Otherwise, the node is unsafe.

Let the SLICE of node f be

$$f = \left(\sum_{i=1}^{n} f_1^i \right) 1 + \left(\sum_{i=1}^{n} f_0^i \right) 0 \tag{3.1}$$

where $i = 1, \ldots, n$. f_l^i is a Boolean logic l equation of signal strength i, where l is either 0 or 1. We have the following lemma to claim the node f is safe.

Lemma 3.4: Given the SLICE of node f in (3.1), if $f_1^i \, f_0^i = 0$ for all $i = 1, \ldots, n$, then the node is safe.

□

Proof: Since $f_1^i \, f_0^i = 0$ for all $i = 1, \ldots, n$, logic 1 and logic 0 of each signal strength i cannot reach node f at the same time for all possible i. Thus, node f is safe.

□

Let

$$F_\Delta = \prod_{i=1}^{n} \overline{f_1^i} \prod_{i=1}^{n} \overline{f_0^i}$$

The definition of F_Δ implies that if there is no conducting paths to node z from the external nodes, then this node is in high impedance state. Note that the signal with the larger strength can always override the signal with the lesser strength. Thus, let F_1 (F_0) be the Boolean logic 1 (0) equation which describes the logic state 1 (0) among different signal strengths. Now we have the following Theorem.

Generalized Theorem 3.1: Given the SLICE of a node f in (3.1), if all the product terms do not disappear and $\overline{F_0 + F_\Delta} = F_1$, then all the nodes present in f's SLICE are safe.

□

Proof: Because $\overline{F_0 + F_\Delta} = F_1$ at node f, logic 1 and logic 0 will not reach node f. This implies $f_1^i \, f_0^i = 0$, for all $i = 1, \ldots, n$. Hence, node f is safe. Suppose that there exists an unsafe node, say node x, in f's SLICE. Since this node is unsafe, both logic 1 and logic 0 of some strength i may reach this x node at some moment. Because no product term disappears, there is a path from node x to node f. Thus, the unsafe state of node x may reach node f through this path. Hence node f is unsafe, and a contradiction.

□

Based on the above theorem, Rule 2 in Section 3.3.1 can be generalized to handle general static MOS circuits.

Generalized Rule 2: Given the SLICE of node f in (3.1), if $\overline{F_0 + F_\Delta} = F_1$, that is

$$\overline{\sum_{i,j=1,\, i\neq 1}^{n} (f_1^i + f_0^j) + F_\Delta} = f_1^1,$$

$$\overline{\sum_{i,j=1,\, i\neq 2}^{n} (f_1^i + f_0^j) + F_\Delta} = f_1^2,$$

$$, \cdots ,$$

$$\overline{\sum_{i,j=1,\, i\neq n}^{n} (f_1^i + f_0^j) + F_\Delta} = f_1^n,$$

then node f is safe; otherwise node f is unsafe. Furthermore, if no product term disappears in f's SLICE, then all the nodes present in the SLICE of f are safe. Also, if $F_\Delta = 0$, then the one Boolean equation F_1 itself is sufficient to represent the functionality of the node f.

□

Justification: Justified from Generalized Theorem 3.1.

□

Note that when n is 1, the Generalized Theorem 3.1 and Generalized Rule 2 degenerate to Theorem 3.1, and Rule 2, respectively.

3.5 Extraction of Dynamic Circuits

It has been shown that the static MOS circuits can be extracted to obtain their Boolean behavior. This has not yet been shown for dynamic circuits. In this section, a rule for dynamic circuit design style is presented. However, it still will not be considered for general dynamic circuits. General dynamic circuits allow nodes with different capacitances. Thus, they may have different node strengths in each node. In the following rule, however, the Boolean behavior of dynamic circuits is realized by the recognition of the circuit pattern.

Rule 4: Let the SLICE of node z be $z = \bar{\phi}\, 1 + g\phi\, 0$, where ϕ is a clock input, node z has a bigger capacitance value than other nodes in the group, and g is a Boolean expression. Then the Boolean equation for node z is $z_1 = \bar{g}$, $z_0 = g$, and $z_\Delta = 0$.

□

Justification: If it is known that ϕ is a clock input, and final SLICE for node z can be simplified to the form as shown in Rule 4, then the circuit is a dynamic circuit. This is clear from the fact that the circuit pattern is a dynamic circuit design style. When ϕ is 0, the node z is charged to logic 1. While ϕ is 1, the node z may remain logic 1 due to node capacitance or become logic 0 through a conducting path from ground. Thus, high impedance state can never happen in node z, and the Boolean equation z_0 is g, and z_1 is $\bar{g}$.

□

Example 3.10: The circuit in Figure 3.11 is a dynamic NOR gate.

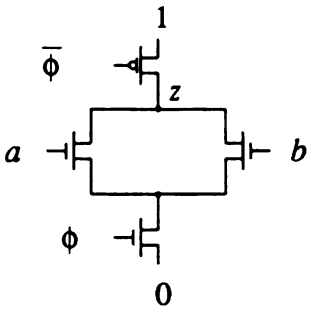


Figure 3.11 Dynamic NOR gate

The SLICE of node z is $\bar{\phi}\,1 + a\,\phi\,0 + b\,\phi\,0 = \bar{\phi}\,1 + (a+b)\,\phi\,0$. From Rule 4, we have $z_1 = \overline{(a+b)}$ which is truly a NOR gate.

□

Several rules have been proposed to extract the Boolean behavior of different design styles of MOS circuits. The circuit class considered is static MOS circuits plus some dynamic circuits. Furthermore, the electrical safe property of a node being extracted in the circuit is examined. In reality, the Generalized Rule 2 will not be applied often, and it is mainly of theoretical interest. Therefore, it is usually sufficient to extract the circuits with every proposed rule except Generalized Rule 2. These rules are used by a complete algorithm for extracting the functionality of a complex transistor group. This extraction algorithm will be introduced in the following chapter.

Chapter 4

Functionality Extraction Procedure

4.1 Functionality Extraction Algorithm for a Transistor Group

In the previous chapter, several rules to extract the Boolean behavior of a transistor group are presented. An extraction algorithm which guides the application of these rules is presented in this chapter. Two things must be taken into consideration during the extraction. First, we must make the extraction process efficient and second, we should be able to locate the error region when extraction fails due to design errors.

For the first consideration, the most important thing is to try to reduce the time complexity of the Boolean comparison which may be encountered in the application of Rule 2. The Boolean comparison is known to be an NP-hard problem [GaJo79]. Thus, to perform the Boolean comparison in a practical way, the number of variables in the Boolean expression cannot be large. Usually the number of variables involved in a transistor group, or a DCC, is small, thus the Boolean comparison complexity cannot dominate the whole extraction process. However, for most DCCs this complexity can be further reduced done by extraction through junction nodes. In this way, the problem of Boolean comparison can be broken down to several smaller subproblems. Thus, the number of Boolean variables involved in each subproblem becomes smaller, and the whole functionality extraction process can be expedited.

Through junction nodes, incorrect design also can be uncovered and located. Since only in these nodes can improper Boolean behavior occur, if we can make sure all the junction nodes in a DCC are safe, then the transistor group being verified is safe. Thus, design errors can be easily identified and located by examining the junction nodes in the transistor group. Therefore, through junction nodes, our two concerns regarding extraction can be addressed. The following example gives us an intuitive feeling for why junction nodes can assist in the extraction process.

Example 4.1: Consider the following circuit with transmission gates. The transistors in the transmission gates are indexed 1 through 4 to distinguish different transistor instances.

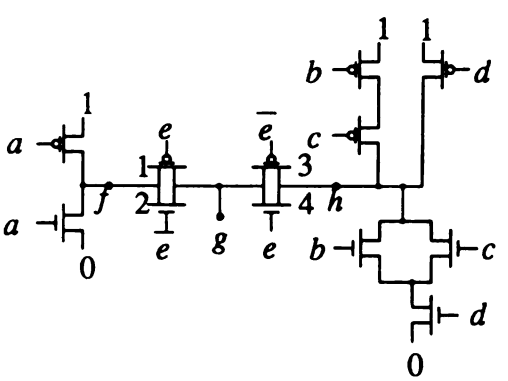


Figure 4.1 Circuit demonstrating divide-and-conquer verification approach

$$g = \overline{e_1}\bar{a}1 + \overline{e_2}\bar{a}1 + e_3\bar{c}\bar{b}1 + e_3\bar{d}1 + e_4\bar{c}\bar{b}1 + e_4\,\bar{d}\,1$$

$$+ \overline{e_1}a0 + \overline{e_2}a0 + e_3bd0 + e_3cd0 + e_4bd0 + e_4cd0$$

$$= (\,\bar{e}\,\bar{a} + e\bar{c}\bar{b} + e\bar{d}\,)1 + (\,\bar{e}a + ebd + ecd\,)0$$

$$g_1 = (\,\bar{e}\,\bar{a} + e\bar{c}\bar{b} + e\bar{d}\,)$$

$$g_0 = (\,\bar{e}a + ebd + ecd\,)$$

$$g_\Delta = (\,e + a\,)(\,\bar{e} + c + b\,)(\,\bar{e} + d\,)(\,e + \bar{a}\,)(\,\bar{e} + \bar{b} + \bar{d}\,)(\,\bar{e} + \bar{c} + \bar{d}\,) = 0$$

The functionality of node g is $(\bar{e}\,\bar{a} + e\bar{c}\bar{b} + e\bar{d})$. Since there is no disappearing product term, the whole circuit is safe. However, the above approach is not a good one because many paths and signal inputs need to be considered. Hence the Boolean comparison complexity is high when applying Rule 2. Also, it may not be easy to identify an error region in the circuit if a design error exists or the circuit is unsafe. A better approach is to use a divide-and-conquer approach.

The better way is to make use of the junction nodes f and h. Consider node f first. $f = \bar{a}1 + a0$. Thus, $f_0 = a$, $f_1 = \bar{a}$, and $f_\Delta = a\bar{a} = 0$. Therefore, $\overline{f_0 + f_\Delta} = f_1$, so the functionality of node f is $\bar{a}$ and it is a safe subcircuit. For node h, $h = \bar{c}\bar{b}1 + \bar{d}1 + bd0 + cd0$. Thus, $h_0 = bd + cd$, $h_1 = \bar{b}\bar{c} + \bar{d}$, and $h_\Delta = (\bar{b} + \bar{d})(\bar{c} + \bar{d})(b + c)\,d = 0$. Therefore, $\overline{h_0 + h_\Delta} = h_1$, and the functionality of node h is $\bar{b}\bar{c} + \bar{d}$ and the subcircuit is safe. Finally, consider node g. $g = \overline{e_1} f + \overline{e_2} f + e_3 h + e_4 h$. After applying Rule 1 and Rule 2, we can find out the functionality of node g which is $\bar{e}f + eh$, and node g is safe. By substituting f and h, we get $g = (\bar{e}\,\bar{a} + e\bar{c}\bar{b} + e\bar{d})$, and the whole circuit is safe. Thus, the Boolean comparison complexity involved has been reduced through the junction nodes when deriving the Boolean behavior of node g through junction nodes f and h.

□

We will first present the extraction algorithm for DCCs which are directed acyclic graphs (DAG) [AhHU74]. Then, an extraction algorithm for general DCCs which allow directed cyclic graphs will be proposed. Usually, the DCCs of most circuits are DAGs. Before giving the algorithm, we will show that only the junction nodes in the DCC need examining to guarantee that the whole circuit is safe. We have the following theorem.

Theorem 4.1: If all the junction nodes in a DCC are safe, then all the nodes in the DCC are safe.

□

Proof: The signal which reaches a non-junction node in the circuit must come through junction nodes or external nodes. Thus, since the signal from a junction node is safe and only one signal can reach the non-junction node from a junction node or an external node, this non-junction node is safe. Hence, all the nodes are safe.

□

4.1.1 Functionality Extraction for Circuits with DAG

The following algorithm FunctionalityExtraction (Algorithm 4.1) is presented to extract the functionality of a specific goal node g in a given transistor group which is a DAG. Before the functionality of the goal node g is extracted, all the transistor directions with respect to this goal node must be determined.

Once the direction of each transistor is determined, then the proposed algorithm can be put to work. If the underlying DCC is a DAG then the algorithm FunctionalityExtraction can be used. The following theorem helps us decide whether a DCC is a DAG or not.

Theorem 4.2: If all the transistors are unidirectional in a transistor group, then the corresponding directed connected component is a DAG. Otherwise it is not a DAG.

□

Proof: Suppose the corresponding directed connected component is not a DAG and there is a directed cycle in the DCC. Let us consider a directed cycle consisting of two edges. Since these two nodes can reach each other, the transistor corresponding to the edges between these two nodes is a bidirectional transistor. This contradicts the assumption that all the transistors are unidirectional.

□

```
Algorithm 4.1: FunctionalityExtraction /* for circuits with directed acyclic graph */
{
    J←{g};  /* g is the goal node, J is the set of current junction nodes */
    S←{ all the external data terminal nodes };
    repeat
        for (each s in S) do
            j = SearchJuncNode(s);
            /* find a junction node j from starting node s */
            J←J+{j};
        end for;
        for (each j in J) do
            JN_j←{ }; /* Initialize JN_j, each junction node j has an associated set JN_j */
        end for;
        for (each s in S) do
            j = LocateJuncNode(s);
            /* Locate the closest junction node j starting from node s */
            JN_j←JN_j+{s}
            /* JN_j is used to keep all the starting nodes s whose junction
            node is j when calling LocateJuncNode(s) */
            SearchAllPaths(s,j);
            /* Find all the paths from node s to node j */
        end for;
        JJ←J;  J←{ };  S←{ }; /* JJ now is the set of current junction nodes */
        for (each j in JJ) do
            if (Depend(j) = TRUE)
            then   /* Node j has no predecessor junction node */
                if (Extract(j) = TRUE)
                /* Try to extract the Boolean equations of node j */
                then    /* Successfully extracted */
                    S←S+{j};
                else
                    Print(Extraction fails for node j);
                end if;
            else
                J←J+{j}; /* Node j cannot be extracted because it has a predecessor
                junction node. Put node j back to J. */
                S←S+JN_j; /* Put starting nodes in JN_j which is associated
                with node j back to S */
            end if;
        end for;
    until (g is not in J);
    /* the Boolean behavior of g is extracted */
}
```

Figure 4.2 Algorithm to perform functionality extraction of circuits with DAG

In this extraction algorithm, the time spent on the Boolean behavior extraction for a group of transistors is mainly for Rule 2 application. In order to reduce the extraction time, the number of variables involved in the Boolean comparison of Rule 2 application must be reduced. This may be achieved by extracting the junction nodes earlier than extracting the goal node. The order of extraction of different junction nodes is important. A junction node cannot be extracted if it has a predecessor junction node which has not been extracted in the DAG. Since this junction node extraction depends on the signal path coming from its predecessor, the predecessor must be extracted prior to this junction node. Thus, for all the junction nodes, there is a precedence relationship among them, and a *junction node precedence graph* (JNPG) can be used to describe the extraction order.

The procedure SearchJuncNode(s) searches for a goal node g starting from a node s. This procedure returns a found node if a junction node or the node g is found. After the first for-loop, all the junction nodes with respect to node g may not be found. However, all the junction nodes eventually will be identified after iterations of repeat-until loops. The procedure LocateJuncNode(s) tries to locate a junction node which is closest to a node s and then return this node. The procedure SearchAllPaths(s,j), starting from a node s, searches all paths that end with a junction node j.

Procedure Depend(j) decides whether a junction node j has any predecessor junction node which has not been extracted. If the node j does have an unextracted predecessor junction node, then it cannot be extracted until all its predecessor junction nodes are extracted. After all signal paths toward a junction node j are obtained, the rules presented in the previous chapter try to derive the corresponding Boolean behavior for this node j. In the procedure Extract(j), when either a junction node or an external node is encountered, Rule 1 is applied to perform the transformation. If a junction node is found with one path connecting to V_{dd} or V_{ss} through an attenuated transistor, Rule 3 is applied to the extract junction node; otherwise, if without attenuated transistor, Rule 2 is

applied. The above procedure is repeated until the goal node g is extracted.

The procedure SearchJuncNode plays a key role in Algorithm 4.1. The detailed algorithm (Algorithm 4.2) for this procedure is shown below.

```
Algorithm 4.2 SearchJuncNode /* Search a junction node j starting from a node s */
/* Before the first call of this algorithm every external node has a signal
associated with it */
/* Initially, internal nodes do not have any signal associated with them */
{
    t = DFS(s); /* Depth first search for a next node t which is a neighboring
    node of a node s */
        while (t ≠ g) /* g is the goal node */
            if ( t in J )
            then
                return({ }); /* t is a junction node already
                identified, return nothing */
            else
                if (SameSignal(t,s))
                then
                    return({ }); /* Nodes t and s are of the same signal,
                    return nothing */
                else
                    if (DifferentSignal(t,s)) /* Nodes t and s are of
                    different signals */
                    then
                        return(j); /* Find a new junction node j */
                    else
                        Signal(t) = s; /* Node t is never reached
                        by DFS before */
                        /* Set signal of node t to s */
                        t = DFS(t); /* Keep searching for
                        next neighboring node from t */
                    endif
                endif
            endif
        endwhile
}
```

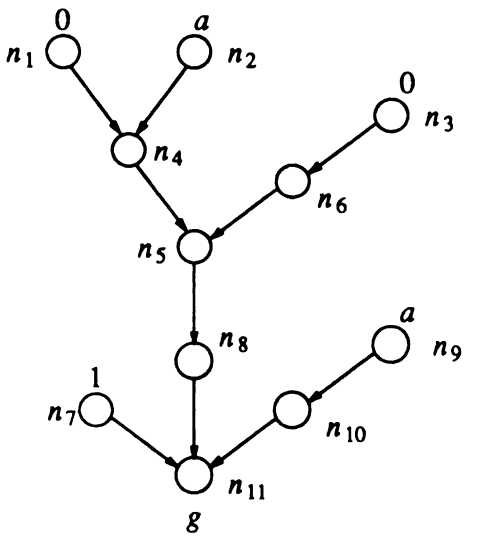
Figure 4.3 Algorithm to search for junction nodes

Algorithm 4.2 SearchJuncNode is the main procedure in Algorithm 4.1. It uses a depth first search algorithm to search for a next neighboring node from the current node, and tries to determine whether this neighboring node is an already identified junction node, a new junction node, a node never searched before, or a node of the same signal as the current node. Whenever the neighboring node is not an identified junction node, if its signal is the same as that of the current node, it is not a junction node. Otherwise, if they are of different signals, a new junction node is identified, or a node never searched before is encountered and the depth first search is continued until the goal node g is reached. Note that the order of junction nodes being discovered depends on the calling sequence of Algorithm 4.2 which is shown in the following example.

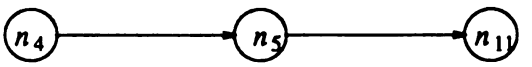
Example 4.1: This example demonstrates Algorithm 4.1 and Algorithm 4.2. The DCC shown in Figure 4.4 (a) has a goal node g.

Before the first repeat-until loop in Algorithm 4.1, $J = \{n_{11}\}$, $S = \{n_1, n_2, n_3, n_7, n_9\}$. During the first iteration of the repeat-until loop of Algorithm 4.1, Algorithm 4.2 is called several times in the first for-loop. Nodes n_4 and n_{11} would be identified as junction nodes after calling SearchJuncNode after the first repeat-until loop. However, node n_5 may or may not be identified as a junction node after the first repeat-until loop, depending on the calling sequence of node s in SearchJuncNode (s). For instance, if the node order is n_3, n_1, n_2 when calling SearchJuncNode in Algorithm 4.1, then node n_5 is not identified as a junction node and node n_4 is. Nevertheless, for another node order n_3, n_2, n_1, both nodes n_5 and n_4 can be identified as junction nodes because both nodes are reached with different signals. No matter what the node order is, all the junction nodes eventually can be identified after several iterations of repeat-until loop.

In this example, at most, three repeat-until loops are needed to identify all the junction nodes and extract their functionality. Suppose junction nodes n_4, n_5 are identified in the first repeat-until loop, and we have $J = \{ n_4, n_5, n_{11} \}$. The LocateJuncNode tries to locate the closest junction node for each node in S set. The closest junction node for node



(a) DCC w.r.t. node g



(b) Corresponding JNPG of the DCC

Figure 4.4 DCC and its JNPG

n_1 or n_2 is n_4. The closest junction node for node n_3 is n_5, and for node n_7 or n_9, it is n_{11}. In the first repeat-until loop, only junction node n_4 can be extracted since it is the only node without predecessor in the JNPG shown. This can be checked through procedure Depend. The signal paths toward n_4 can be decided by SearchAllPaths, and they are $< (n_1,n_4) >$ and $< (n_2, n_4) >$. Before starting the second repeat-until loop, we now have $J = \{n_5, n_{11}\}$, $S = \{n_4, n_3, n_7, n_9\}$ if the extraction of node n_4 is successful. Since node n_5 precedes node n_{11} in the JNPG, junction node n_5, which is the closest junction node for nodes n_4 and n_3, can be extracted during the second repeat-until loop. The paths found toward node n_5 in SearchAllPaths are $< (n_4,n_5) >$ and $< (n_3,n_6), (n_6,n_5) >$. In the beginning of the third repeat-until loop, suppose node n_5 is successfully extracted, then $J = \{n_{11}\}$, $S = \{n_5, n_7, n_9\}$. Node n_{11}, which is the goal node, is

extracted in this iteration.

□

It can be shown that all the junction nodes in the DAG can be identified and examined by the SearchJuncNode called in Algorithm 4.1. Thus, the electrical property of a DCC can also be verified along with the Boolean behavior extraction. The following theorem shows this.

Theorem 4.3: The Algorithm 4.1 FunctionalityExtraction can identify all the junction nodes in a given DCC.

□

Proof: Initially in Algorithm 4.1, J, which is the junction node set, contains only g, the goal node, and set S consists of all the external nodes. In the first repeat-until loop, the junction nodes, which are created because of different signals from the external nodes, can be identified by the procedure SearchJuncNode. From the second repeat-until loop on, S set contains junction nodes, which have been successfully extracted, and external nodes which have not yet contributed to create a junction node in the previous repeat-until loop. Of course, J set contains the node g and junction nodes which are newly found or are not successfully extracted in the previous repeat-until loop due to the violation of the junction node precedence relation. The repeat-until loop is repeated until junction node g in J is successfully extracted.

From the definition of junction node, we know that the signal of a junction node comes either from external nodes or from junction nodes. Therefore, the way to find junction nodes in the repeat-until loop satisfies the definition of junction node. Since the underlying transistor group is a DAG, the path from each external node or each junction node toward the goal node is uniquely determined and only one such path is possible. Thus, whenever each junction node is identified, the way to find all the possible signal paths entering this junction node is uniquely determined and there is no other way to get

these signal paths. Hence, through Algorithm 4.1, all the junction nodes in the transistor group can be identified.

□

Example 4.2: Consider the following MOS circuit of a transistor group, the goal node is *z*. This circuit uses many design styles including pass transistor, transmission gate, pseudo-nMOS, and complementary CMOS. The precedence graph of the junction nodes is also shown.

Initially, $J=\{z\}$ and S is the set of all the external nodes which are data terminals. In the first iteration of the repeat-until loop, we may have $J=\{A, B, C, D, k, z\}$. Note that node k may appear in either the first or the second iteration of the repeat-until loop depending on the junction node finding algorithm SearchJuncNode. By applying Rule 2, the respective functionalities of node A and node C are

$$A = \overline{a}\,\overline{b} \quad C = \overline{s}\,\overline{t}\,\overline{q}\,\overline{p} + \overline{u}\,\overline{v}\,\overline{q}\,\overline{p} + \overline{r}\,\overline{p}$$

Both node A and node C are safe. By applying Rule 3, we have

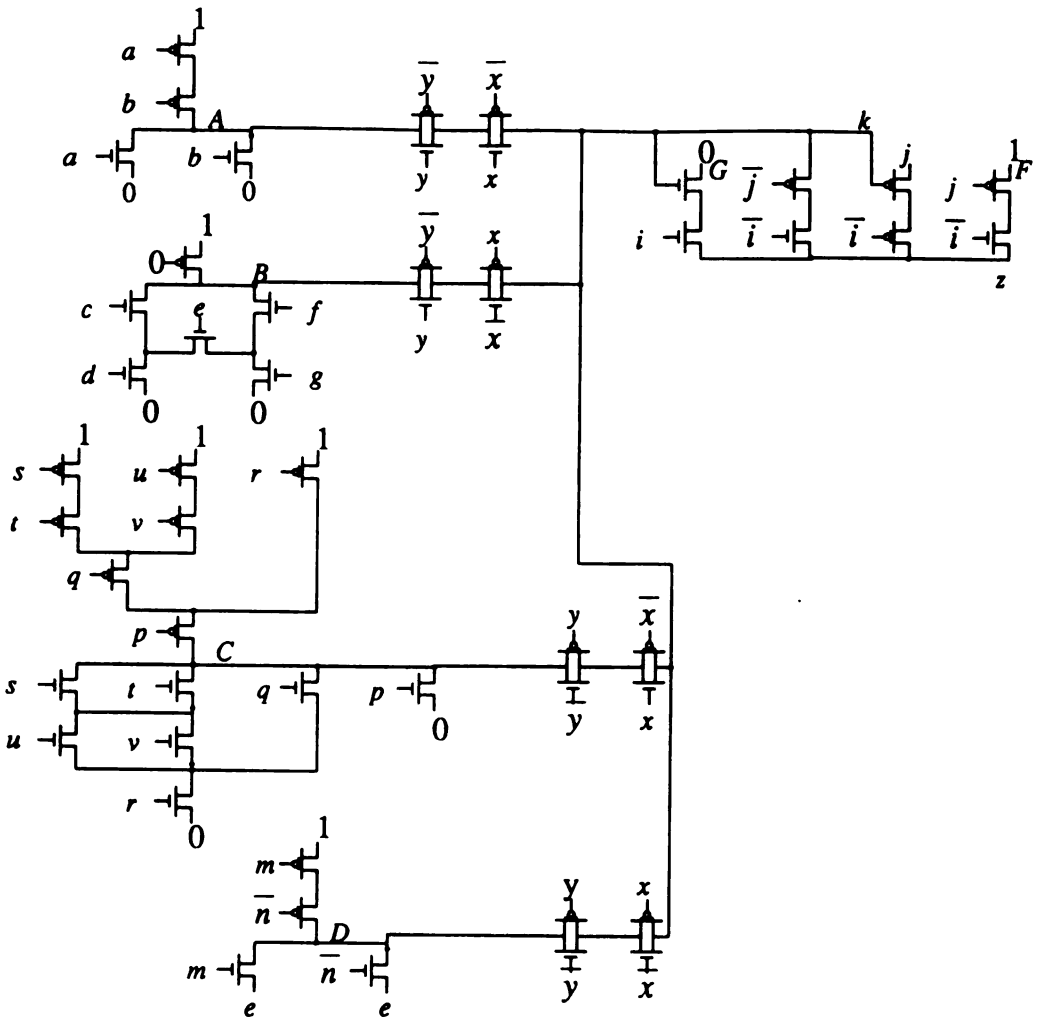
$$B = \overline{(c\,d + f\,g + c\,e\,g + f\,e\,d)} \quad .$$

The functionality of node D is extracted and found to be safe by applying Rule 1 and Rule 2. $D = e + \overline{m}\,n$

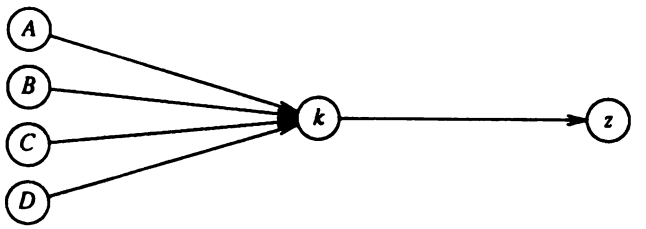
At the end of the first repeat-until iteration, we obtain $J=\{k, z\}$ and $S=\{A, B, C, D, F, G, j\}$. Since k has predecessors in the junction node precedence graph, node k cannot be extracted in the first iteration. In the second iteration of the repeat-until loop, we can extract the node k, which is the predecessor junction node of node z, by applying Rule 1 and Rule 2.

$$k = (x\,y\,A + \overline{x}\,y\,B + x\,\overline{y}\,C + \overline{x}\,\overline{y}\,D)$$

Node k is safe, J becomes $\{z\}$, and S is $\{k, F, G, j\}$. In the third iteration, the goal node z is extracted and safe by applying Rule 1 and Rule 2. Thus $z = \overline{i}\,k + i\,j\,\overline{k} + \overline{i}\,\overline{j}$.



(a) Circuit with many design styles



(b) Corresponding JNPG

Figure 4.5 Circuit to be extracted

□

It is important to analyze Algorithm 4.1 so that the time complexity of this algorithm can be fully understood. We first examine each procedure called inside Algorithm 4.1. Considering the procedure SearchJuncNode, from Algorithm 4.2 we know that its worst case complexity is bounded by the longest path length from an external node toward the goal node. Procedure LocateJuncNode is still bounded by the longest path length in the worst case since it tries to locate the closest junction node along a path, on which there may be more than one junction node, toward the goal node. The procedure SearchAllPaths searches for all the paths between two nodes. It is theoretically possible that an exponential number of paths may occur between two nodes in a DCC. However, in reality, for almost all circuits, the number of paths tends to be small if only a portion of DCC is considered each time.

The procedure Depend examines whether a junction node has a predecessor junction node. This can be done by traveling from each junction node found toward the goal node. If during the traveling, a junction node is passed by, then this passed-by junction node cannot be extracted. Therefore, the time complexity for this procedure is bounded by the longest path length and the number of junction nodes in the current junction node set J. As mentioned, the computation time of the procedure Extract is mainly spent in Rule 2. However, by the extraction of junction nodes in a divide-and-conquer fashion, the time spent in the Boolean comparison may be significantly reduced.

When Algorithm 4.1 is finished, each of the procedures SearchJuncNode, LocateJuncNode, and SearchAllPaths is called $O(n)$ times in the worst case, where n is the number of nodes in the circuit. The procedure Extract is called i times in the worst case where i is the number of junction nodes in the circuit, while the procedure Depend is called $O(i)$ times in the worst case. Thus, we know that the time complexity in Algorithm 4.1 is dominated by the procedures Extract and SearchAllPaths. However, through the junction nodes, the time spent in both procedures can be greatly reduced,

since each time only a portion of a DCC is considered.

4.1.2 Functionality Extraction for General Circuits

If a DCC contains cycles because of bidirectional transistors, then this DCC is not a DAG. We will present an algorithm to handle DCCs which are directed cyclic graphs. This algorithm is based on the Algorithm 4.1 and relies on finding *strongly connected components* [AhHU74] of directed cyclic graphs. A strongly connected component (SCC) [AhHU74] of a directed graph is a maximal set of vertices in which there is a path from any one vertex in the set to any other vertex in the set. By finding out the SCCs of a DCC, it is possible to obtained a new DAG called *pseudo DAG* (PDAG) which is the original DCC with some of its SCCs replaced by a node called *pseudo nodes*. The following algorithm is used to extract the functionality of a DCC having cycles.

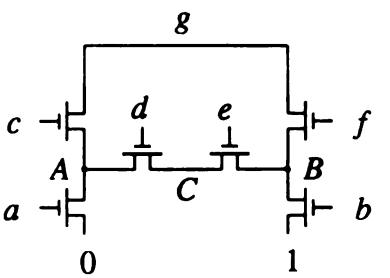
Algorithm 4.3: FunctionalityExtraction /* for a general directed graph */

1. Find all the SCCs for the given DCC and construct its PDAG. The algorithm to find SCCs can be found in [AhHU74].
2. If different signals enter a certain SCC, then all the nodes in this SCC are junction nodes, and this SCC stands as a *pseudo junction node* in the corresponding PDAG. Otherwise, this SCC is just a pseudo node in the PDAG.
3. Use the Algorithm 4.1 FunctionalityExtraction to perform the functionality extraction on this PDAG.
4. When extracting the pseudo junction node, first identify the junction nodes in the pseudo junction node, if any, which leave for other nodes outside the pseudo junction node. Then all the signal paths toward these junction nodes from the paths entering this pseudo junction node must be figured out. Thus the functionality of these junction nodes can be derived.

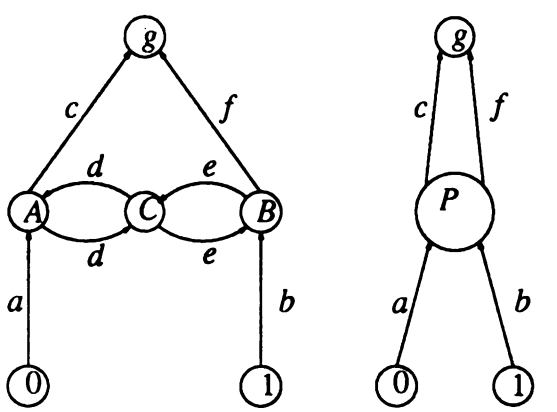
Figure 4.6 Algorithm for the functionality extraction of a general directed graph

Note that the time complexity to find SCCs in a DCC is $O(\max(n,e))$ where n is the number of nodes in DCC, and e the number of edges [AhHU74]. Therefore, the introduction of SCC does not significantly increase complexity in Algorithm 4.3. Also, note that extra time is needed to do the path searching in step 4 of Algorithm 4.3. As mentioned, path searching may encounter numerous paths. However, in practice, the number of paths is limited.

Example 4.3: In the circuit of Figure 4.7, there are two bidirectional transistors with gate terminals d and e.



(a) Circuit with bidirectional transistors



(b) Corresponding DCC (c) PDAG

Figure 4.7 Circuit with cycles in its DCC

In the corresponding DCC, an SCC consisting of nodes A, B, and C can be identified. The corresponding PDAG is shown and node P is the pseudo junction node in this PDAG. From the PDAG the SLICE of node g can be derived. Nodes A and B in the SCC (pseudo junction node) are leaving for node g which is the only node that can be reached from the SCC. Thus, all the signals entering nodes A and B outside the SCC have to be discovered. We have $A = a0 + deb1$, and $B = b1 + eda0$, Also, $g = cA + fB$. Hence, the Boolean equations of node g can be determined.

□

4.2 Functionality Aggregation

After the functionality extraction of each transistor group is done, the final functionality of the whole circuit can be derived by the aggregation of the individual Boolean behavior of each transistor group.

4.2.1 Aggregating Functionalities of Groups

Most of the aggregation may be achieved through the substitution of the Boolean equations of extracted transistor groups. To do this substitution, we have to understand the relationships among the groups. These relationships can be revealed through the input-output relationships among transistor groups. These input-output relationships are reflected in the gate inputs of gate nodes. If the output of one transistor group is the input of the gate node of another transistor group, then we say that the former transistor group precedes the latter one. Thus, we can define a directed graph called *transistor group precedence graph* (TGPG), where each vertex in TGPG is a transistor group. A directed edge is defined from u to v, if u and v are vertices of TGPG and u precedes v. From this TGPG, the order of aggregation by substitution among transistor groups can be determined. We may use techniques such as topological sort [AhHU83] to decide the order of

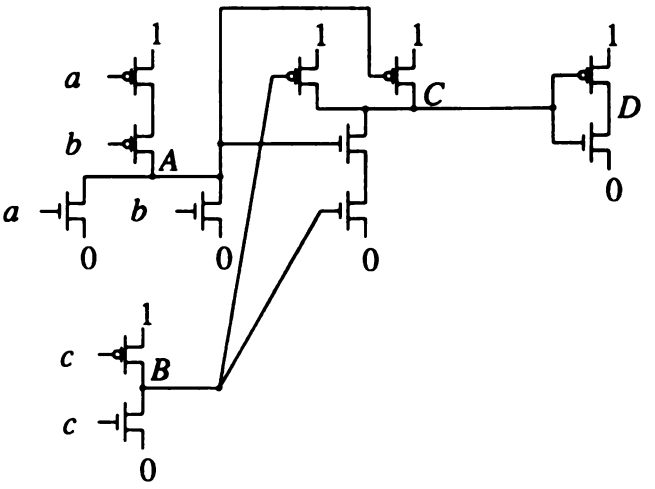
aggregation.

However, we may encounter feedback loops which presents the sequential behavior of the circuits. Thus, in the corresponding TGPG, directed cycles may certainly occur. For a TGPG with cycles, we may first try to find the strongly connected components (SCCs) of a TGPG. We can then try to identify the sequential behavior of each SCC found. Some sequential behavior, such as RS-latch, may not be properly reflected by a Boolean expression derivation. Extra knowledge about some sequential circuit properties may be needed during the functionality aggregation to correctly identify circuit behavior. Once all the SCCs are found and replaced by new vertices representing these SCCs in the TGPG, the appropriate order of aggregation by substitution can be performed using topological sort to obtain the overall circuit functionality. However, for still other circuits, we need to explore more properties among different transistor groups to determine their Boolean behavior. Usually this comes from different circuit design styles, and more rules may be required to recognize these styles. In the following subsection, this kind of circuits will be discussed.

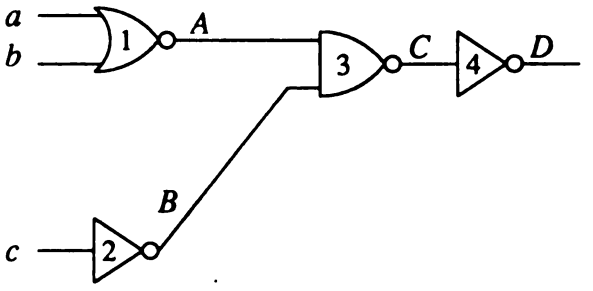
If an input to a transistor group coming from another group is described by only a single Boolean equation of Boolean logic 1, the aggregation by substitution is straightforward. It may be considered as a design error for static MOS circuits when the input is a Boolean logic Δ equation. However, if the input contains a Boolean logic 1 or 0 equation, as well as a Boolean logic Δ equation, then we may use the Boolean logic 1 equation to continue the substitution and also to properly reflect a design error warning message. Another way is to use another state, called unknown state, to represent the input and to continue the substitution. However, this method will generate a number of unknown states, so that the final aggregated Boolean equations may not be useful for functionality comparison.

The following example shows how to obtain an aggregated functionality by substitution.

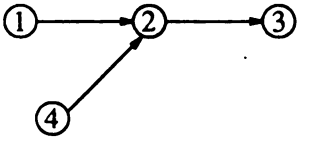
Example 4.4: Figure 4.8 shows a circuits at both the switch level and gate level, respectively.



(a) Switch level circuit with four transistor groups



(b) Corresponding gate level representation



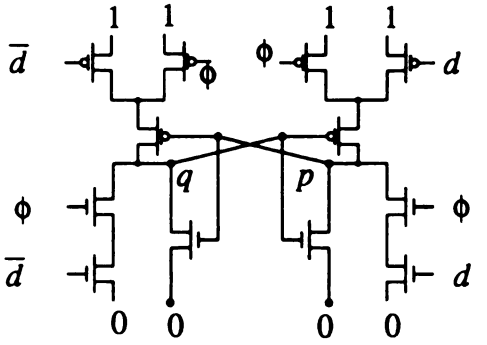
(c) Corresponding TGPG

Figure 4.8 Circuit demonstrating functionality aggregation

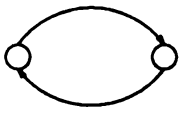
The circuit has four transistor groups and the corresponding TGPG is also given. For each group, the Boolean behavior is extracted respectively as $D = \overline{C}$, $C = \overline{AB}$, $A = \overline{a + b}$, and $B = \overline{c}$. The order of aggregating the Boolean behavior is determined by the precedence in the TGPG. We can see that gate 3 must be substituted before gate 4 is processed. We first have to substitute $C = \overline{AB} = \overline{\overline{(a + b)}\,\overline{c}}$. Then the second substitution is $D = \overline{C}$, so $D = \overline{\overline{\overline{(a + b)}\,\overline{c}}} = \overline{(a + b)}\,\overline{c}$, which is the final aggregated Boolean behavior.

□

Example 4.5: The circuit in Figure 4.9 is a CMOS D flip-flop, a memory element. The functionality of node q is to be found.



(a) CMOS D flip-flop



(b) Corresponding TGPG

Figure 4.9 Aggregated circuit

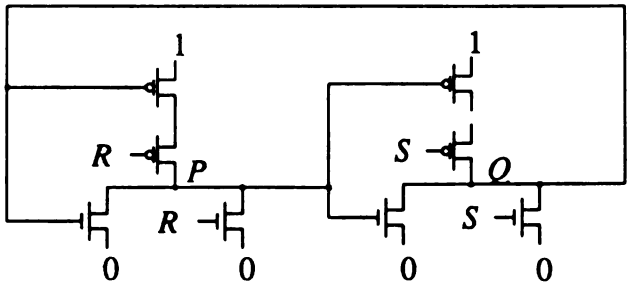
This circuit has two transistor groups. The corresponding TGPG has a cycle, so the circuit has sequential behavior. Once the logic function of each group is recognized, in this case $q = \overline{\phi\,\overline{d} + p}$ and $p = \overline{\phi\,d + q}$, we can recognize the function of the binary storage

cell by substituting one signal name and making the other a function of time. Thus, we have $q = \overline{\phi\,\bar{d} + p} = (\bar{\phi} + d)\,\bar{p}$. Therefore, we obtain

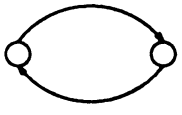
$$q_{n+1} = (\bar{\phi} + d)(\phi\, d + q_n) = \bar{\phi}\, q_n + d\,\phi.$$

□

Example 4.6: The circuit of RS-latch is shown in Figure 4.10.



(a) RS-latch



(b) Corresponding TGPG

Figure 4.10 Circuit needs extra knowledge when aggregating

There are two transistor groups. The outputs of each group are $P = \overline{Q + R}$, and $Q = \overline{P + S}$, respectively. However, the output of Q is not simply obtained by substitution to get $Q_{n+1} = \overline{\overline{(Q_n + R)} + S}$, which is obviously not the Boolean behavior we are familiar with for RS-latch. Thus, more knowledge is needed during aggregation to identify the behavior of some sequential circuits.

□

4.2.2 Extraction Rule Among Groups

Some MOS design styles may be defined within different transistor groups. During the functionality aggregation, some other rules may be needed to uncover those styles and to derive the Boolean behavior of each style from the relationships among transistor groups.

There is a tightly coupled design style called DCVS logic [HGDT84], which can be identified by recognizing the relationship between two different transistor groups. Therefore, some extra rule is required to find this special relationship among transistor groups after the functionality extraction of each group is done. The DCVS design is achieved by converting all p-type transistors of a fully complementary CMOS gate to n-type transistors and by adding two cross-coupled p-type transistors. It is always possible to obtain both true and false values of the original logic function. The following rule identifies two transistor groups which use DCVS design style.

Rule 4: Let a SCC from a TGPG consist of only two transistor groups, and let the output nodes of these groups be x and y, respectively. Given the SLICEs of node x and y, if $x = \bar{y}1 + f0$, $y = \bar{x}1 + g0$, and $f = \bar{g}$, then $x = \bar{f}$, and $y = \bar{g} = f$.

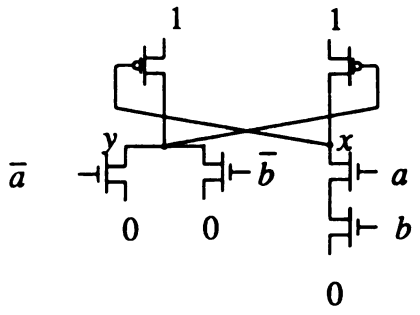
□

Justification: Since $f = \bar{g}$, the conducting paths from V_{ss} to nodes x and y complement each other. Thus, nodes x and y cannot both have logic 0 at the same time. However, whenever one node gets logic 0, say node x, the other node y will get a logic 1 because node x turns on the p transistor connecting node y and V_{dd}. Therefore, the Boolean expressions of nodes x and y are complementary, and $x = \bar{f}$, $y = f$.

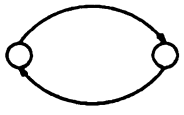
□

Example 4.7: Consider in Figure 4.11 the DCVS logic with two transistor groups whose TGPG is a directed cycle. SLICEs of nodes x and y, respectively, are

$$x = \bar{y}\ 1 + a\ b\ 0$$



(a) Circuit using DCVS logic



(b) Corresponding TGPG

Figure 4.11 Circuit with DCVS design style

$$y = \bar{x}\,1 + (\bar{a} + \bar{b})\,0$$

Since $\bar{a} + \bar{b} = \overline{a\,b}$, we have $x = \overline{a\,b} = \bar{a} + \bar{b}$ and $y = \overline{\bar{a} + \bar{b}} = a\,b$.

□

In general, if still other design styles are defined among different transistor groups, more rules will be required to describe each style. During the final functionality aggregation phase, these rules then have to be applied to derive the final functionality.

4.3 Conclusion

A functionality extraction method for static MOS circuits and part of dynamic MOS circuits has been proposed. From the signal flow point of view, the approach can unify the functionality extraction of different design styles. Rules are presented to extract the Boolean behavior as well as to perform electrical safety checking at certain special

nodes, namely junction nodes. An algorithm is introduced to identify these special nodes and guide the rules for performing the functionality extraction. The extraction process is fast, because the number of rules is small and they are only applied to a small portion of a transistor group each time. Furthermore, it is easy to locate the design error, because the error region is confined to a small region by the algorithm during extraction. The final overall functionality of the circuit can be obtained by transistor group aggregation. Extra knowledge may be required during the aggregation to understand the behavior of some circuits which cannot be well represented by Boolean expressions such as RS-latch. Moreover, more rules may be needed for design styles which are defined within different transistor groups.

Most of this method has been implemented in C language. Since the algorithm tries to derive the functionality of a group in a divide-and-conquer fashion, the time required for applying rules and finding signal paths is insignificant. The functionality extraction time in the experiment is shown in Table 4.1. Most of the extraction time is spent in applying Rule 2. This rule has to perform Boolean comparison, and the performance may be improved by using better comparison heuristics.

circuits	no. of transistors	extraction time (secs.)
alu1	110	0.7
cnt3	130	0.5
onepulse	186	0.8
LFSR	428	2.2
regtrs	672	3.6
alu8	2280	12.0

Table 4.1: Extraction time for different circuits on SUN3/280

Chapter 5

Signal Flow Analysis in Timing Verification

5.1 Introduction

As pointed out before, the SLICE can be used as a design representation. The circuit designers may use SLICEs to represent and describe both the circuit connectivity and signal flow direction of MOS circuits being designed. SLICE representation is better than traditional graph-based representation such as transistor net lists, since graph-based representation provides only connectivity information but no functional information such as signal flow direction. In Section 5.2, we demonstrate how to perform timing verification for the circuits expressed in SLICE. In the subsequent sections, we apply the SLICE representation and junction node concept to the false path problem which occurs in a transistor group during timing verification. An algorithm to derive the correct signal paths is presented first, and then a heuristic is proposed to accelerate the correct signal path finding process.

5.2 Timing Verification on SLICE Representation

In this section we demonstrate the ability of SLICE to efficiently perform timing verification of MOS circuits. A number of circuit simulation programs such as SPICE [Nage75], have been developed to measure the timing performance of integrated circuits.

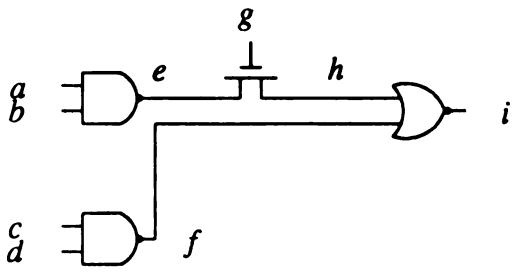
These programs give a more accurate result, but they are very time consuming. Thus, they are impractical for today's VLSI circuits which may contain hundreds of thousands of transistors. The switch level model is then proposed to estimate the circuit performance, which is much faster, but at the expense of less accuracy [Joup87a, Oust85]. The switch level model considers each transistor in the circuit as a perfect switch with a certain value of resistance, and each node in the circuit associated with a certain value of capacitance. The resistance and capacitance of a circuit is usually estimated by the circuit designer before the actual layout is performed. The timing estimation of the circuit can then be easily computed based on the given RC information [RuPe83, Oust85]. A more accurate timing measurement can be obtained by feeding back more accurate RC information after the circuit layout is done. Obviously, timing analysis at the switch level is an important step in the circuit design, and, thus, an appropriate switch level circuit representation is essential to the efficiency of the timing verification process.

To perform timing verification at the switch level, the representation of a MOS circuit schematic should provide both the circuit connectivity and the signal flow information needed to calculate the circuit delay time [Joup87a, Joup87a, Oust85]. Path enumeration and critical path analysis on a circuit schematic are two basic procedures involved in the timing verification process. These procedures are required to identify the signal flow direction of transistors and to extract the circuit stages of the circuit being verified. Since some fixed logic values of circuit inputs are allowed in timing verification, the timing verifier should have the ability to handle logic simulation [Hitc82, Joup87b, Oust85].

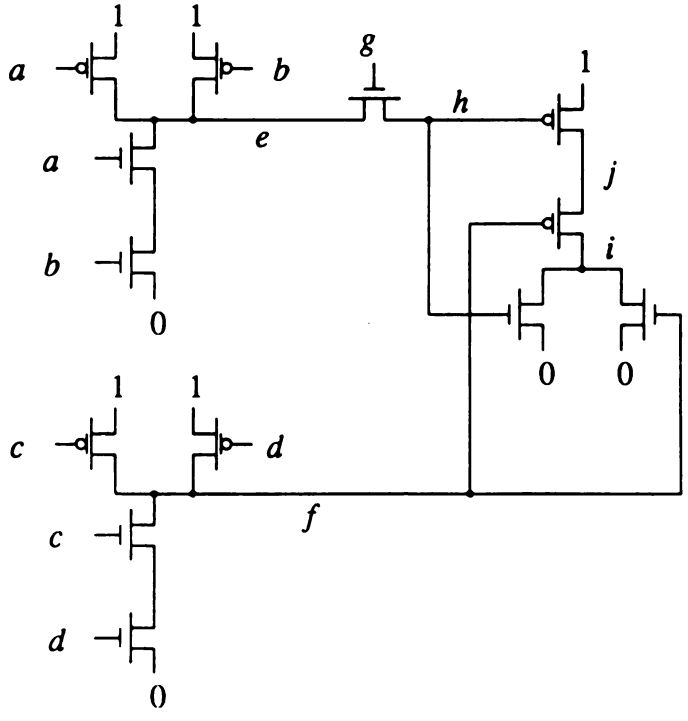
In timing verification, the path enumeration technique tries to find all the possible signal paths in the circuit. It starts from the output and traces back until an input is reached. Now consider a MOS circuit represented by a set of SLICEs. Given an output node, say node z, we try to find all paths coming from external inputs to the node z. First, find the SLICE, in the form of (2.1), of the current node z. For the right hand side of this

expression, if node d_j in (2.1) is a junction node, then d_j is in the path. Otherwise, d_j is an external node and the corresponding nodes of the gate terminal nodes of g_{ji} in (2.1) are in the path. This procedure is repeatedly applied to each newly found node on the possible paths until the external input nodes are reached.

Let us consider the example in Figure 5.1.



(a) Gate level network



(b) Circuit schematic

Figure 5.1 Schematic of a CMOS circuit and its corresponding gate level network

For instance, the SLICEs of the four nodes in Figure 5.1 are

$$i = \bar{h}\,\bar{f}\,1 + h0 + f0 \tag{5.1}$$

$$h = ge \tag{5.2}$$

$$e = \bar{a}\,1 + \bar{b}\,1 + ab0 \tag{5.3}$$

$$f = \bar{c}\,1 + \bar{d}\,1 + cd0 \tag{5.4}$$

The above four expressions completely specify the signal flow and connectivity of the circuit shown in Figure 5.1. The primary output node *i*, expressed in (5.1), is defined in terms of *h*. Then, we have *h* in terms of *e*, and *e* in terms of *a* and *b* which are external input nodes. Two paths (*a*,*e*,*h*,*i*) and (*b*,*e*,*h*,*i*) are then obtained. Two other paths from nodes *c* and *d* through *f* to *i* can be similarly found.

For critical path analysis, one has to start from some external inputs at a given instant. Then one has to follow certain paths until an output node is reached. During this path following process, the timing delay along each stage is recorded and updated. Thus, the time instance at which the output node is reached from the worst case path delay can be obtained.

Based on the SLICE representation, the critical path can be found as follows. First, find a SLICE which has external input signals specified on its right hand side. Then calculate the delay for the left hand side node of this SLICE, and record the delay time for this node. Considering this left hand side node as an input, we have to repeat this procedure until the desired output node is reached. Thus, the delay time for a path from external inputs to an observed output node can be obtained. However, this path may not be the critical path. We have to consider all possible SLICEs which are affected by the external input signals. Then, the path with the longest delay time is the critical path. Apparently, the critical path finding described above is a depth first search with the longest delay time pruning [Oust85].

In the example of Figure 5.1, we want to measure the delay at output *i*. We begin applying input signals at *a* and *b*. We start from (5.3) because *a* and *b* are in (5.3). Since the left hand side of (5.3) is in (5.2), we proceed to (5.2), and then to (5.1) in which we want to measure the delay time for node *i*.

In timing verification, before the delay is calculated, we have to extract and identify stages which are usually logic gates in the transistor network. Once all stages have been extracted, the delay time of each stage can be computed. By summing up the delay times of different stages, the delay of a signal path is obtained. Note that the SLICE representation explicitly identifies stages in terms of junction nodes or gate nodes. If a graph-based circuit representation is used, additional effort must be spent to identify the circuit stages.

The direction of the signal flow in a circuit is important for performing timing verification. Again, the graph-based circuit representation provides no signal flow information. Let's examine the example in Figure 5.2 which is a two-bit shifter with two inputs and two outputs.

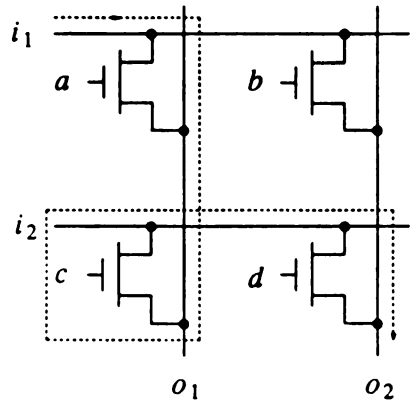


Figure 5.2 2-bit shifter

The SLICE representation for the circuit in Figure 5.2 is shown below.

$$o_1 = a\, i_1 + c\, i_2$$

$$o_2 = b\, i_1 + d\, i_2$$

Note that a and c cannot be logic 1 at the same time, nor can b and d. As demonstrated in Figure 5.2, the dash path from input i_1 to output o_2 may be considered as a possible path if using graph-based representation. However, this false path never exists in a real shift register. To avoid finding such a false path, other information is needed besides the representation of the circuit connectivity. Usually this information is provided by the designer in terms of signal flow direction. In the SLICE representation, the direction of signal flow of all transistors is clearly described. Thus, no false path will be derived.

During timing verification, we may also have to perform a case analysis in which the fixed logic values of some inputs are given. It is then necessary to propagate these inputs to all possible subsequent stages. Therefore, the timing verifier should be able to perform logic simulation when needed. With SLICEs, the switch level simulation can be done systematically no matter what kind of transistor logic style is employed. The logic simulation capability of the SLICE representation will be demonstrated in the next chapter.

5.3 False Signal Path Problems in Timing Verification

Timing verification [Hitch82, Joup87a, Oust85] is a widely used technique to examine the circuit performance in the VLSI design community. Timing verifiers try to find the longest propagation delay path (critical path) in the circuit, and check whether it meets the circuit design requirement.

It is known that for some MOS circuits there are problems for timing verifiers in identifying the correct signal paths. Therefore, false critical paths may be reported. There are two kinds of false path problems in timing verification. The first kind occurs in a general digital network among transistor groups [BrIy88, BMCM87, DuYG89]. The

other one happens within a transistor group [Joup87b, Oust85]. Both kinds of problems are caused by a lack of proper functional information during timing verification.

Let us examine the first kind of false path problem. Suppose each gate of the circuit in Figure 5.3 has unit delay 1. This circuit has three transistor groups. A timing verifier without considering functional information may report the longest delay of node f as being three units. However, when c is logic 1, f gets logic 1 after one unit, and when c is logic 0, f gets logic 0 after two units. Thus, because some gates are always blocked by signal c, the longest delay is actually two units. Hence, if the functional information such as gate types and external inputs is not taken into consideration, a timing verifier may falsely report a path delay which is too pessimistic.

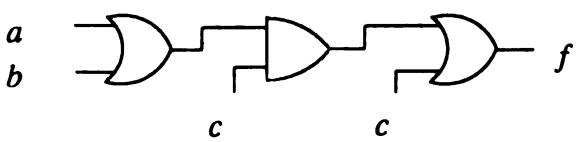


Figure 5.3 Circuit with false path problem

For another kind of false path problem, let us take another example. Consider the multiplexer circuit in Figure 5.4 which itself is a transistor group. We want to know the longest delay of node g_2. The longest delay path may be considered as $< \bar{b}, c, \bar{c} >$ or $< b, c, \bar{c} >$ which is expressed in terms of l-path. However, this is not true, since these two signal paths are never activated when considering gate terminal node inputs c and $\bar{c}$, which block paths. The correct delay path of node g_2 actually is , $< \bar{a} >$ or $< a >$. Thus, again, if the functional information of the circuit is not taken into consideration by the timing verifiers, false signal paths may be reported and result in an incorrect timing estimation.

We will consider the false path problem of the second kind in this research. For this problem, methods have been proposed to derive the correct signal flow direction of all the transistors in a transistor group of MOS circuits [Joup87b, Oust85]. One solution

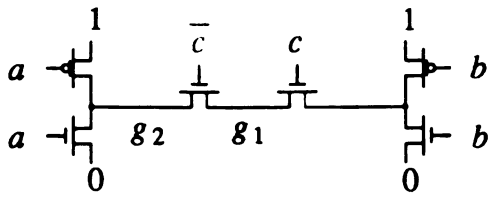


Figure 5.4 Multiplexer circuit

requires users to tag all the unidirectional pass transistors, which are difficult to handle, with direction flags [Oust85]. Obviously, this approach involves error-prone user input. Another approach tries to derive the correct signal flow direction through a set of rules [Joup87b]. However, since the number of rules is not small, the application of the rules is complicated.

A method to find the correct signal flow direction of MOS transistors in a transistor group during timing verification is proposed. The proposed method takes into account information which is usually ignored by timing verifiers. Thus, it is able to identify the correct signal flow directions for the problematic circuits during timing verification. Distinguishing different kinds of nodes in a transistor network and making use of transistor gate terminal node names (functional names) can aid the derivation of signal flow direction for MOS transistors. Using both transistor network connectivity information and functional information, a more effective signal flow analysis method is developed for finding the correct signal flow direction. Although the method is able to find correct signal paths, it can sometimes encounter time consuming pathfinding To speed up the running time of pathfinding, a heuristic is proposed to perform the signal path derivation. This heuristic does not guarantee to obtain all the correct signal paths. However, most of the time it has been found to work well.

5.4 Method to Identify Correct Signal Flow Direction

In this section, a method is proposed to find the correct signal flow direction in timing verification. When the timing constraints of a MOS transistor network are to be verified, the circuit is first partitioned into transistor groups. For each transistor group, the signal flow direction of each transistor is to be determined. Since signals always flow from external nodes and are blocked by the goal nodes (either output nodes or gate nodes), the possible signal paths can be derived from paths starting from external nodes and ending at the goal nodes. The signal flow direction of each transistor in a transistor group can be determined by the following procedure. This procedure uses functional information to eliminate the impossible signal paths.

Algorithm 5.1 Algorithm to eliminate false signal paths in a transistor group

For each goal node in the transistor group

1. Signal flow directions are determined by all the directed signal paths starting from external nodes and ending at the goal node.
2. Each time a path is found, the product term of the corresponding l-path must be examined. If it disappears after Boolean simplification, then the direction of the signal path is set to each transistor until some transistor blocks the signal path. This signal path is then discarded and not considered. If the signal path does not disappear, then this path is a valid one.

Note that during Boolean simplification, all the intermediate variables in the gate product term may need to be replaced by the signal names of the external nodes so that the Boolean operation can be fully exploited.

If only the signals from the external nodes are considered to affect signal paths, the above procedure can guarantee to find the correct signal paths and set the correct signal

flow direction of each transistor in a group. The time complexity of Algorithm 5.1 is proportional to the number of signal paths in the transistor group. In the worst case, the possible signal paths, which usually contain many false paths, may be exponential in the number of nodes in a transistor group. Still, the correct signal paths can eventually be determined by using Algorithm 5.1.

Consider the circuit in Figure 5.4. Let the goal node be node g_2. The SLICE of g_2 is $\bar{a}1 + a0 + \bar{c}c\bar{b}1 + \bar{c}cb0 = \bar{a}1 + a0$. Product terms $\bar{c}c\bar{b}$ and $\bar{c}cb$ disappear, thus the corresponding two signal paths never occur. The directions of the transistors with gate terminal nodes a and b are set, but not the transistor with gate terminal node c. Now let the goal node be node g_1, and $\bar{c}\,\bar{a}1 + \bar{c}a0 + c\bar{b}1 + cb0$ is the SLICE of g_1. The four signal paths in the SLICE are all valid and set the direction of all the transistors accordingly.

Usually timing verifiers do not consider the functional information of the underlying circuit. Among many signal paths, verifiers use a delay calculator to compute the signal delay for each path, and then find the longest propagation delay path. This is an expensive way to obtain the critical path since the delay calculation is costly, and moreover, the paths found may be incapable of occurring in the actual circuit. However, by using functional information we can eliminate paths that never occur in the circuits before performing the delay calculation. Therefore, the total computation cost of finding a critical path can be reduced, and the reporting of false paths can be avoided.

In most circuits, a transistor group has a limited number of correct signal paths. When many signal paths are reported, especially in a group with multiple goal nodes, it is very likely that false paths which never occur in the actual circuits are also reported. Thus, in the case where the number of paths reported exceeds a certain threshold number, it is believed that false paths may be reported among the superfluous number of paths. Therefore, we may abort the signal path derivation method of Algorithm 5.1, and instead use a heuristic method to accelerate the signal path finding process. The heuristic does not guarantee to find all the correct signal paths, but it speeds up the identification of

correct signal paths in most cases. Therefore, the heuristic will not be used unless there is an indication that false paths may be reported. It would also be better to notify users that the timing verifier is running the heuristic, so that users may double check the signal flow direction for themselves. In the following section, a heuristic is proposed to quickly eliminate false paths in the timing verification of a transistor group.

5.5 Heuristic to Find Signal Path Direction

The idea behind the proposed heuristic is the observation of the relationships between external nodes and goal nodes. A signal starts propagating from external nodes and searches for goal nodes. If a goal node or gate node is reached, it may not be necessary to propagate away from the reached goal node or gate node since the output nodes or gate nodes seem likely to block the signal paths. Also, when a junction node is formed, it seems likely to behave as a new signal source. Therefore, an identified junction node may be treated as a new external node providing a signal source. By using the heuristic, we may in advance eliminate many impossible signal paths before we use functional information to eliminate false paths. The algorithm of the proposed heuristic is as follows.

Algorithm 5.2 Heuristic to find correct signal paths in a transistor group

For each goal node in the transistor group

1. Signal directions are determined by all directed signal paths starting from external nodes to a goal node.
2. During the search, if a gate node is encountered, then the path searching from the current external node stops at this gate node. It is not necessary to keep searching for the goal node. The search continues from the remaining external nodes, if any.
3. During the search, if a junction node is encountered, then the path searching from the current external node stops at this junction node. It is not necessary to keep

searching for the goal node. The search continues from the remaining external nodes, if any.

4. When searching for a particular goal node, if other goal nodes are encountered, then the path searching from the current external node stops at this goal node. There is no need to keep on searching for the goal node. The search continues from the remaining external nodes, if any.
5. If a junction node is identified after the search from all the external nodes is finished, this junction node can then become a *pseudo external node*. We treat each pseudo external node as an external node and repeat steps 1-5. Thus we may reset the transistor direction which has been previously set.
6. When steps 1-5 can not be applied any more, then we search for all the directed paths formed by the transistor direction set in steps 1-5. Each time a path is found, the product term of the corresponding 1-path must be examined. If it disappears after Boolean simplification, then the direction of the signal path is set to each transistor until some transistor blocks the signal path. This signal path is then discarded and is not considered. If the signal path does not disappear, then this path is a valid one.

This heuristic tends to assume that all the transistors in the circuit are unidirectional. Thus, the heuristic may fail when there are true bidirectional transistors in the circuit. An exponential number of signal paths may still be encountered in the circuits when steps 1-5 of Algorithm 5.2 are applied. However, if the circuit contains many junction nodes, goal nodes, or gate nodes, direction of every transistor may be determined by the complexity of the number of transistors in the circuit. This is because the heuristic is likely to set the direction of each transistor in the circuit as unidirectional. Step 6 evaluates all the possible paths from the transistor direction determined from steps 1-5. Though the number of paths evaluated may still be exponential, it is much less than it would be

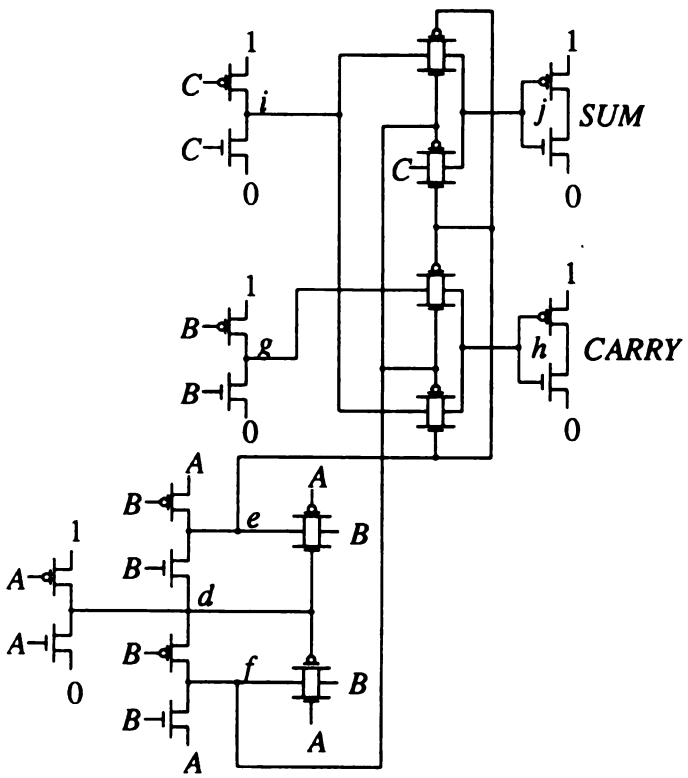
without using the heuristic.

5.6 Experiments and Discussion

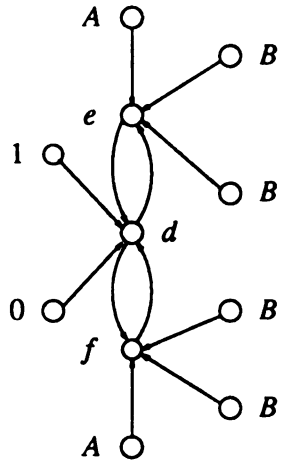
In this section, we will show how to apply the proposed heuristic to different circuits whose correct signal paths are difficult for timing verifiers to determine. A threshold value for the number of paths found is used to decide whether the heuristic should be run or not. A useful guideline is, the more goal nodes in a transistor group, the larger the threshold value. The heuristic can quickly identify the correct signal paths in most of the problematic circuits.

Consider the full adder circuit shown in Figure 5.5. There are four transistor groups in this circuit. The heuristic is applied when the total number of paths exceeds 10, the threshold value, for the group containing goal nodes *e* and *f*. Step 2 is applied when the heuristic encounters node *d* which is a gate node. Then consider the node *d*, which is a pseudo external node, as an external node. Step 5 is applied. Without the heuristic, node *e* or *f* each has eight signal paths going toward them and three of them are false paths. These false paths can be eliminated using functional information. However, with the heuristic the number of signal paths is correctly reduced to five for each node before functional information is used to eliminate false paths, as shown in the directed group graph.

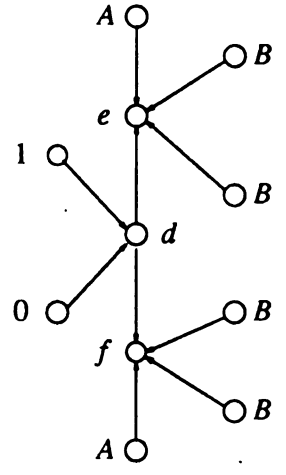
Now let us again examine the barrel shifter circuit. A 3-bit barrel shifter with output nodes *o* 1, *o* 2, and *o* 3 is shown in Figure 5.6. Without using the heuristic, this circuit contains too many false paths. However, applying steps 4 and 5 of the heuristic, the correct signal flow direction is easily obtained as shown in the directed group graph. Though the heuristic is working in this example, it may have trouble when the three nodes in the circuit cannot be easily identified as output nodes by the timing verifier.



(a) Full adder circuit

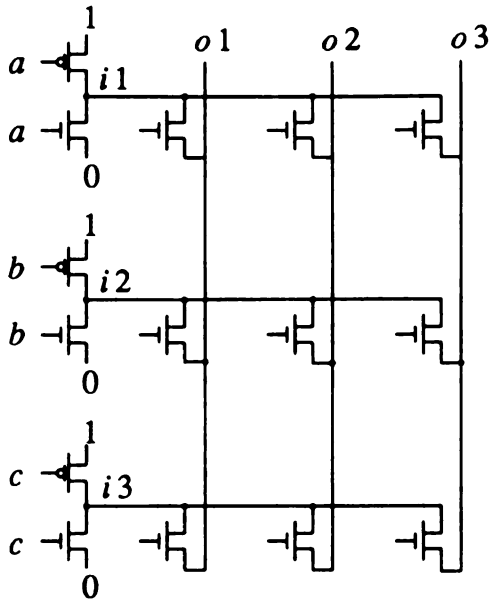


(b) Path finding without heuristic

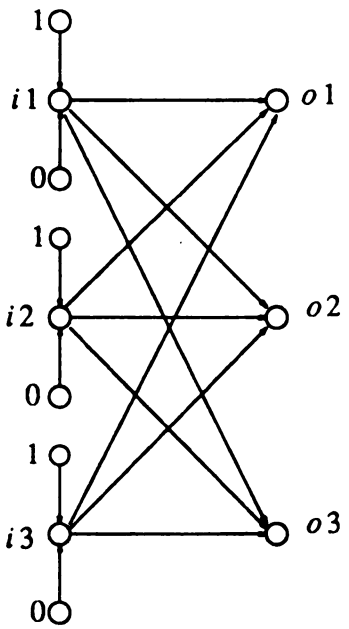


(c) Path finding with heuristic

Figure 5.5 Full adder and its directed group graphs

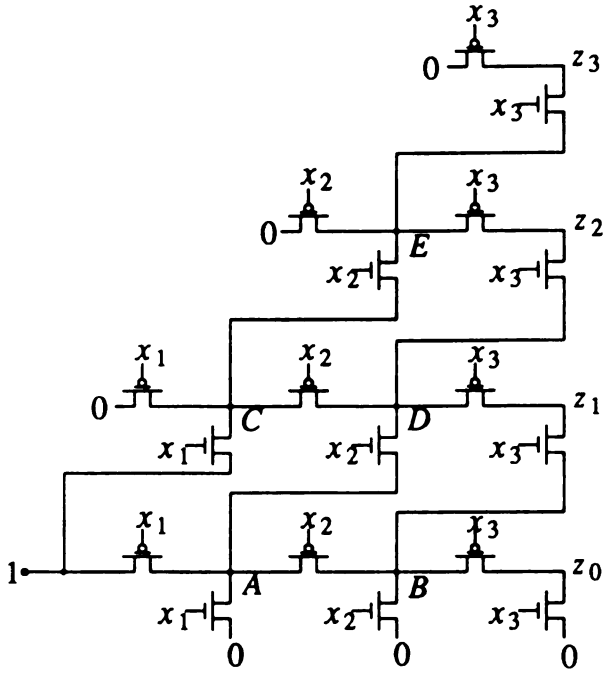


(a) 3-bit barrel shifter circuit

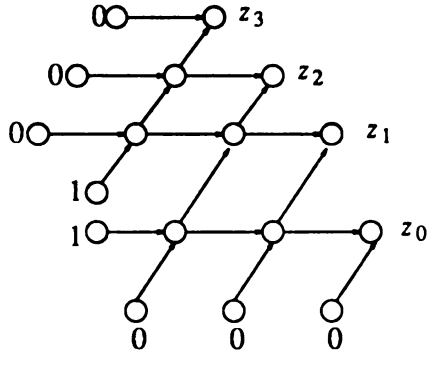


(b) Directed group graph derived from the heuristic

Figure 5.6 3-bit barrel shifter and its directed group graph



(a) Tally circuit



(b) Directed group graph derived from the heuristic

Figure 5.7 Tally circuit and its corresponding directed group graph

Thus, as can be seen, the success of the heuristic is not guaranteed.

For the circuit shown in Figure 5.7, this tally circuit definitely has many possible paths. After applying steps 3, 4 and 5 of the heuristic repeatedly, surprisingly, the signal flow direction can be determined correctly as is shown.

The proposed method only considers the signal flow direction derivation from the external signal source point of view. For circuits using static design, the proposed method can find the correct signal flow direction. However, the signal flow direction of a circuit depends not only on external signal sources, but also on node properties in the circuit, such as node capacitance. To more precisely identify the correct signal flow directions, the node properties of the circuit must be considered. For the circuit shown in Figure 5.8, the transistor with gate terminal node c is determined to have no direction from the proposed method. However, if the capacitance of node $C2$ is much greater than that of node $C1$, then the signal flow direction is from node $C2$ to node $C1$ and that transistor becomes a unidirectional transistor. Consideration of node capacitance when finding the correct signal flow direction is not covered in this work.

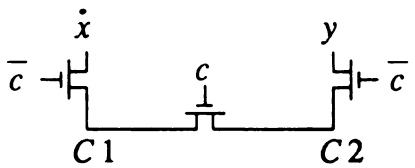


Figure 5.8 Circuit with different capacitances at nodes $C1$ and $C2$

In summary, a method to find the correct signal flow direction of MOS transistors in timing verification is presented. Since both the connectivity information and the functional information of transistor networks are considered, this method guarantees to find all the correct possible signal flow directions in the circuits from the signal flow point of view. A heuristic is presented to quickly determine signal flow direction. This heuristic does not ensure that every signal path can be correctly found. However, in most cases, it facilitates rapid reduction of the false signal paths and faster identification of the correct

signal paths.

Chapter 6

Logic Simulation with SLICE

6.1 Logic Simulation at the Switch Level

Logic simulation is a commonly used circuit verification method in the VLSI design community. Logic simulation may be performed at different circuit levels such as the functional block level, gate level, or switch level. Since SLICE is a circuit representation at the switch level, it would be necessary to understand how a switch level logic simulation can be performed on the circuits expressed in SLICE. Since SLICE describes both connectivity and functionality of the circuit being verified, circuit designers may specify the SLICE representation from the functionality (i.e., signal flow) point of view. Thus, each SLICE represents a functional block partitioned from the original circuit. How to partition the whole circuit really depends on the designer's approach. However, each SLICE is a subcircuit of a transistor group, i.e. it may be a transistor group or a portion of a transistor group. Because we are only dealing with logic behavior, we are not concerned with the timing behavior of the circuit. We may assume each SLICE expression has a unit delay. Once we have a set of SLICEs to represent a circuit being implemented, logic simulation at the switch level may be performed on this set of SLICEs. An event driven scheme is usually used in logic simulation. Therefore, we may use event driven simulation among different functional blocks, each of which is a SLICE representation.

However, our emphasis is not on event driven schemes, which are a common technique, but on the logic value evaluation of each SLICE during the simulation.

The logic states of four-valued Boolean algebra [Haye82] will be considered first. These states are sufficient to simulate static MOS circuits with no ratioed logic such as pseudo nMOS design. The lattice structure which describes four-valued Boolean algebra is shown in Figure 6.1 (a). Let 1, 0, Δ and U be the logic levels corresponding to Boolean logic 1, Boolean logic 0, indeterminate value, and some types of faults (such as short circuits), respectively. For every SLICE expressed in (2.1), the logic value of the left hand side node, say node z, is to be determined. Then when the logic levels of each signal name in the right hand side of node z's SLICE are available, the new logic level of node z can be evaluated. Note that only the logic level of the nodes in the left hand side of a SLIDE needs to be evaluated, and it is not necessary to be concerned with the logic level of other nodes in the circuit.

6.2. Logic Evaluation of SLICE in Simulation

In this section, we will show how to evaluate the logic level of a node expressed in SLICE. g_i mentioned in the following is interpreted as the logic level for the gate terminal node of the i-th transistor, and d represents the logic level of a node. The logic level of g_i denotes the logic level (negation logic level) of a gate terminal node of n-type (p-type) transistor. Let $L = 0$, $H = 1$, and $Y = U$.

The following rules are defined to evaluate the logic level of product terms for a node z in the SLICE of (2.1).

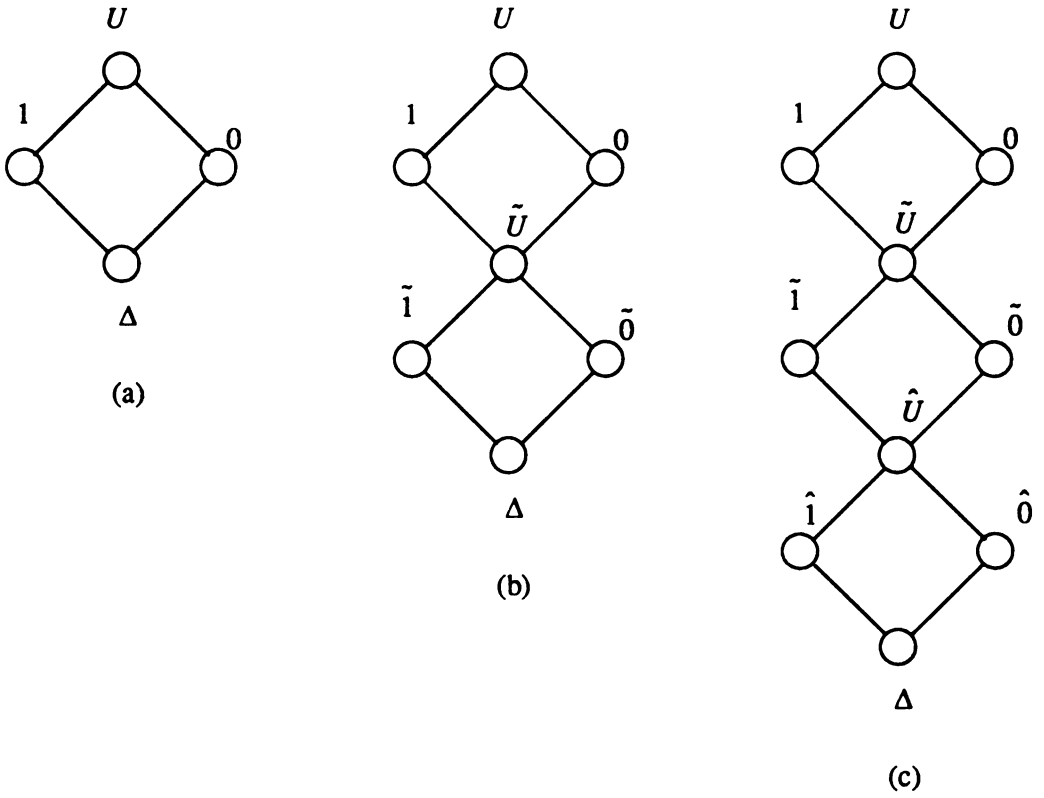


Figure 6.1 Lattice structures of different logic levels

Rule 1: $(\prod_i g_i)\, d = L$ if $g_i = H$ for all i and $d = L$ (6.1)

Rule 2: $(\prod_i g_i)\, d = H$ if $g_i = H$ for all i and $d = H$ (6.2)

Rule 3: $(\prod_i g_i)\, d = \Delta$ if there exists an l such that $g_l = L$ or $d = \Delta$ (6.3)

Rule 4: $(\prod_i g_i)\, d = Y$ if there exists an l such that $g_l = Y$ or $d = Y$ (6.4)

To evaluate the summation of different logic levels of a node in (2.1), the following rules are defined. They are based on the lattice structure of four elements in Figure 6.1 (a).

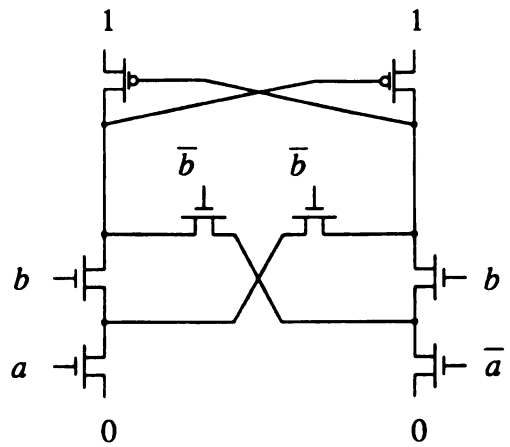


Figure 6.2 DCVS XOR/XNOR gate

$$\Delta+\Delta=\Delta \qquad \Delta+1=1 \qquad 1+0=Y \qquad Y+1=Y$$
$$\Delta+0=0 \qquad 0+0=0 \qquad Y+Y=Y \qquad Y+0=Y$$

The following example demonstrates the switch level logic simulation on a DCVS [HGDT84] XOR/XNOR gate. The SLICEs of the circuit shown in Figure 6.2 are described below.

$$y=(\bar{z})_8 1+b_1(\bar{a})_2 0+b_1(\bar{b})_5 z+(\bar{b})_6 a_4 0+(\bar{b})_6 b_3 z$$

$$y=(\bar{z})_8 1+b_1(\bar{a})_2 0+(\bar{b})_6 a_4 0$$

$$z=(\bar{y})_7 1+b_3 a_4 0+b_3(\bar{b})_6 y+(\bar{b})_5(\bar{a})_2 0+(\bar{b})_5 b_1 y$$

$$z=(\bar{y})_7 1+b_3 a_4 0+(\bar{b})_5(\bar{a})_2 0$$

Nodes y and z are the outputs of NOR and XOR, respectively, and can be characterized by the above expressions. If node a and node b are set to logic 1 and logic 0, respectively, then logic states of nodes y and z can be evaluated according to the above rules as follows:

$$y=\bar{z}1+110+\bar{1}00=\Delta+0+\Delta=0$$

$$z = \bar{y}\,1 + 100 + 010 = \bar{0}1 + \Delta + \Delta = 1$$

To simulate a static circuit with ratioed design, seven logic levels {0, 1, Δ, U, $\tilde{0}$, $\tilde{1}$, $\tilde{U}$} [Haye82] are needed, where $\tilde{0}$, $\tilde{1}$, and $\tilde{U}$ represent weak signals. The four rules used to evaluate the product term in the above can be easily modified to accommodate simulation by reducing the signal strength when the signal is traversing through a transistor of different strength. The L, H, and Y in the above four rules have to extend to allow L in {0, $\tilde{0}$}, H in {1, $\tilde{1}$}, and Y in {U, $\tilde{U}$}. Also the rules for summation operations are now based on the partial ordering lattice structure with seven elements shown in Figure 6.1 (b).

For MOS circuits with a dynamic logic design style, ten logic levels [Haye82] are sufficient to simulate the circuit in practice. The corresponding partial ordering lattice structure which describes ten-valued logic is shown in Figure 6.1 (c), where { $\hat{0}$, $\hat{1}$, $\hat{U}$ } are stored charge signals. To perform logic simulation for the circuits based on the ten-valued logic, the above product term rules can still be used, but extra checks are needed for all the dynamic nodes. Since ten-valued logic is used, there are only two possible strengths of a node. If a static signal (signal from external nodes not from stored charge) reaches a dynamic node, then the logic level of this node is decided by this signal; otherwise, the logic level of this node is its previous logic state in the form of its stored charge. For dynamic circuits, L in the above rules is modified to include $\hat{0}$. Similarly, H includes $\hat{1}$, and Y includes $\hat{U}$. The summation operations of product terms for the circuits with ten logic levels can be performed according to the lattice structure in Figure 6.1 (c).

For the example in Figure 5.1, node h is a dynamic node and suppose its present logic level is 1. If a, b, c, and d are all set to logic 1, and g is set to logic 0, the logic value of node i can be obtained as follows. From (5.3), we have

$$e = \bar{a}1 + \bar{b}1 + ab0 = \Delta + \Delta + 0 = 0$$

From (5.2), we get

$$h = ge = 00 = \Delta$$

Since h is a dynamic node, h will keep its previous logic value when no other static signal reaches it. So the logic state of h is still 1. From (5.4), we have

$$f = \bar{c}\,1 + \bar{d}\,1 + cd0 = \Delta + \Delta + 0 = 0$$

From (5.1), we obtain

$$i = \bar{f}\,\bar{h}1 + h0 + f0 = \Delta + 0 + \Delta = 0$$

Thus, we obtained the logic value of node i, which is logic 0, from a given set of input logic values.

We can see from what has been presented that the proposed switch level logic simulation has the following advantages when performing simulation on the circuit expressed in SLICE representation.

(1) Static partition of the circuit is explicitly represented in SLICE expressions.

(2) Only simple logic operations are involved in logic value evaluation. Thus, no dynamic partition [Brya84] is needed to simulate different input patterns. Since the signal flow direction is provided, far fewer nodes need to be evaluated for the steady state logic level compared to other simulation approaches. Thus, simulation time can be saved.

(3) The SLICE representation unifies complementary gate logic and pass transistor logic, so it is suitable for mixed level simulation, including both gate level and switch level, as shown in Figure 5.1 (a).

It is easy to extend the proposed simulator to cover different strengths of MOS transistors in the circuits. However, it is difficult to cover different strengths of nodes in the circuits, since charge sharing between nodes must be handled. Though the method proposed may not be suitable for handling theoretical MOS circuits which allow any

number of logic strengths, it is practical enough to deal with real MOS circuits of different design styles including ratioed and dynamic logics.

Chapter 7

Conclusion and Future Work

7.1 Summary

We have presented the verification methods for MOS circuits represented in SLICE expression. In this chapter we will summarize the work and the contribution we have made, with discussion of possible future research.

Chapter 1 introduced the motivation behind this research. We realized that an effective switch level circuit representation would facilitate the circuit verification process. Some background about the switch level model of MOS circuits and circuit verification methods was reviewed. Previous work and the problems in different verification methods were also briefly mentioned.

The switch level MOS circuit model employed in this research was presented in Chapter 2. A graph representation describing a switch level MOS circuit was presented. Then we introduced a directed graph to describe the behavior of MOS circuits. Based on the directed graph, we proposed a switch level circuit representation called Structured LogIcal Circuit Expression (SLICE) which carries both the structural and functional information of a MOS circuit. The representation enabled us to develop more effective and efficient circuit verification methods. We then examined the derivation of Boolean behavior from a SLICE. This derivation is important for functional verification.

Preliminary results of this research were presented in [LiNi88].

In Chapter 3, several rules were proposed to extract the Boolean behavior of a transistor group represented by a SLICE. These rules consider the electrical safety of the circuit, and examine whether a circuit is electrically safe or not. Thus, the unknown logic states caused by design error can be discovered. Different design styles of MOS circuits can be handled well by these extraction rules. In particular, the extraction of gate logic and pass transistor logic can be unified through the proposed rules. We also discussed the extraction rules for the design style of general static MOS circuits. An extraction rule for some dynamic circuits was also presented.

Functionality extraction algorithms to guide the application of the extraction rules were presented in Chapter 4. The extraction algorithm for a transistor group with a directed acyclic graph was proposed first. In order to have an efficient extraction process, we introduced the novel concept of the junction nodes in a circuit. Through these junction nodes and the divide-and-conquer approach, the time complexity of the Boolean comparison, which is an important operation in the extraction process, can be greatly reduced. Then we expanded the extraction algorithm to handle a transistor group with a general directed graph, using the idea of the strongly connected component. After showing how to extract the functionality of each group in the circuit, we then presented a method to aggregate the functionality of each group and obtain the final overall functionality of the whole circuit. An additional extraction rule was presented to help accomplish the process of aggregation. Experiment result implementing the extraction process were also shown. The initial concept of functionality extraction was presented in [LiNi89].

In chapter 5, we explained how to perform timing verification on the circuits represented in SLICE. Then the false signal path problems in timing verification were introduced. We only considered the problem of finding correct signal paths within a transistor group. Distinguishing different types of nodes in the transistor network and

making use of transistor gate terminal node names facilitated the derivation of the correct signal paths. Thus, an effective method to determine the correct signal paths was developed. Although the method is able to find correct signal paths, it sometimes encounters time consuming signal pathfinding. To speed up the pathfinding computation time, a heuristic was proposed to perform signal direction derivation and signal pathfinding. This heuristic does not guarantee to obtain all the correct signal paths. However, most of the time it has been found to work well.

In Chapter 6, we demonstrated the advantages of performing logic simulation on the circuit in SLICE representation. Since not all the logic states of every node in the circuit need to be known, and only those nodes that are on the left hand side of the SLICE need to be simulated, the simulation time can be reduced. Rules to evaluate logic states during simulation were also presented.

In summary, a circuit representation, SLICE, was proposed, which carries both the structural and functional information of the circuit. Effective verification methods which take advantage of the information provided by SLICE were then developed. These methods include functionality extraction, timing verification, correct signal pathfinding, and logic simulation. Thus, the functional and timing behavior of a MOS circuit can be completely verified through its SLICE representation.

7.2 Future Work

The work we have presented is, of course, not exhaustive. There is still much room for future research. Here are some possible areas for further study.

(1) Verification methods for general MOS circuits

In this research, we have limited our discussion to static MOS circuits and some dynamic MOS circuits. We do not consider those dynamic MOS circuits which allow charge sharing phenomena in the circuits. However, some circuit designs do make use of

charge sharing techniques to accomplish some desired functional behavior. Thus, extending our verification methods to cover all the dynamic MOS circuits is important to make the proposed methods complete. When we want to consider the general MOS circuits, then, each node in the circuit should have different node capacitances. Each node capacitance in the circuit holds a charge which can contribute its logic state to some other nodes. Thus, every node in the circuit is considered as a possible signal source. An efficient algorithm to identify useful signal paths that affect the goal nodes is the main object of research in the extension of the proposed verification methods for dynamic circuits.

(2) Efficient methods to automatically generate SLICE

It is assumed that the SLICE representation describing the behavior of a circuit is available before circuit verification is performed. When the SLICE is not available, we suggested that a path searching algorithm like depth first search might be used to identify all the signal paths and then generate the SLICE expression. However, with such an approach, the time complexity may be high. Thus, a more efficient path searching method is needed. One possible approach is to add some rules, such as those suggested in [Joup87b] to guide the path searching algorithm by reducing the search space. In this way, a more efficient search method may be obtained to automatically generate the SLICE representation of the circuit.

(3) Fault diagnosis using SLICE representation

Uncovering the faults in the circuits is crucial in VLSI design. In order to do this, test generation is required. Test generation generates a set of test patterns capable of sensitizing the signal paths. From these paths, a faulty circuit can be distinguished from a fault-free circuit. Signal path analysis is needed to generate useful test patterns. Since SLICE representation carries signal flow information, it can assist the signal path analysis and generate appropriate test patterns at the switch level. Test generation techniques at the switch level are not well developed and usually each technique is only

applicable to a special class of circuits [Agra84, RoSh85]. It would be interesting to consider the test generation for the class of static MOS circuits using SLICE representation.

(4) Finding false signal paths among different transistor groups

We have proposed a method to eliminate false signal paths within a transistor group by utilizing the functional information in the circuit. Similarly, we may investigate the false signal path problem among transistor groups by considering the functional information revealed in the SLICE. Since the SLICE is able to describe transistor interconnection of circuits at the switch level, and furthermore, to identify different stages of functional blocks in the circuit such as logic gates, the functional relationships among different groups are embedded in the SLICE. Using this information, we may develop methods to eliminate the false signal paths which occur among different transistor groups in the circuit designed at the switch level.

(5) Application of proposed verification method to digital bipolar circuits

Earlier research did not investigate the need to describe the digital bipolar circuits at the switch level. More recently, this need has been addressed in [HaSa87]. Since bipolar transistors are physically unidirectional devices, and do not have charge sharing phenomena between nodes in the circuit, the switch level model of four-valued lattice structure can well describe the behavior of the bipolar circuits. Because the SLICE representation is suitable for such static circuits as those with four-valued lattice structures, the verification methods we have proposed can be profitably applied to bipolar circuits. In this way, the functional and timing behavior of the bipolar circuits can be understood by incorporating an appropriate timing model of bipolar transistors in the proposed verification methods.

(6) Circuit synthesis through SLICE representation

We have proposed a SLICE representation and investigated different verification methods on the circuits in a SLICE. But we have not discussed the possibility of circuit

synthesis through SLICE representation. Since a SLICE is a design representation, it is possible to manipulate a SLICE with some design constraints to yield several different SLICEs. Thus, we can use a SLICE to accomplish a circuit synthesis. The synthesis is especially good for pass transistor logic design style, because a SLICE can clearly describe the signal flow of the circuit at the switch level. Actually, a SLICE is similar in some ways to a binary decision diagram (BDD) [Aker78], which has been used in synthesis [Brya86]. Hence, it would be interesting to exploit circuit synthesis at the switch level through a SLICE representation.

(7) Incorporating efficient Boolean comparison methods in functionality extraction

We have tried to reduce the Boolean comparison complexity during the functionality extraction process. The Boolean comparison computation time can be further reduced by using a better Boolean comparison algorithm. This will improve the overall performance of the functionality extraction process.

List of References

[Agra84] P. Agrawal, "Test Generation at Switch Level," *Proc. International Conference on Computer Aided Design*, 1984, pp. 128-130.

[AhHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, 1974.

[AhHU83] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *Data Structure and Algorithm*, Reading, MA:Addison-Wesley, 1983.

[Aker78] S.B. Akers, "Binary Decision Diagram," *IEEE Trans. Computers*, Vol. c-27, NO. 6, June 1978, pp. 509-516.

[BaTr80] C. Baker and C. Terman, "Tools for verifying Integrated Circuit Designs," *Lambda Magazine*, 4th Quarter 1980, pp. 22-30.

[BMCM87] J. Benkoski, E.V. Meersch, L. Claesen, and H. De Man, "Efficient Algorithms for Solving the False Path Problem in Timing Verification," *International Conference on Computer Aided Design*, 1987, pp. 44-47.

[BrIy88] D. Brand and V.S. Iyengar, "Timing Analysis Using Functional Analysis," *IEEE Trans. Computers, Oct. 1988, pp. 1309-1314.*

[Brya80] R.E. Bryant, "An Algorithm for MOS Logic Simulation," *LAMBDA Magazine*, Vol. 1, no. 3, 1980, pp. 46-53.

[Brya84] R.E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Trans. Computers*, Feb. 1984, pp. 160-177.

[Brya85] R.E. Bryant, "Symbolic Verification of MOS circuits," *1985 Chapel Hill Conf. VLSI*, 1985, pp. 419-438.

[Brya86] R.E. Bryant, "Graph-based Algorithm for Boolean Function Manipulation," *IEEE Trans. Computer*, Vol. c-35, Aug. 1986, pp. 677-691.

[Brya87] R.E. Bryant, "Boolean Analysis of MOS Circuits," *IEEE Trans. Computer-Aided Design*, Vol. CAD-6, NO. 4, July 1987, pp. 634-649.

[DuYG89] D.H.C. Du, S.H.C. Yen, and S. Ghanta "On the General False Path Problem in Timing Analysis", *26th Design Automation Conference*, 1989, pp. 555-

560.

[EbZa83] C. Eberling and O. Zajicek, "Validating VLSI Circuit Layout by Wirelist Comparison," *Proc. 1983 Int'l Conf. Computer- Aided Design,* Santa Clara, Calif., Sept. 1983, pp. 172-173.

[GaJo79] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York: Freeman, 1979.

[HaSa83] I.N. Hajj and D. Saab, "Symbolic Logic Simulation of MOS Circuits," *Proc. Int. Symp. Circuits Syst.*, 1983 pp. 246-249.

[HaSa87] I.N. Hajj and D. Saab, "Switch Level Logic Simulation of Digital Bipolar Circuits," *Computer-Aided Design*, Vol. CAD 6, NO. 2, March 1987, pp. 251-258.

[Haye82] J.P. Hayes, "A Unified Switching Theory with Applications to VLSI Design," *Proc. IEEE*, 1982, pp. 1140-1151.

[HGDT84] L.G. Heller, W.R. Griffin, J.W. Davis and N.G. Thoma, "Cascode Voltage Switch Logic: A Differential CMOS Logic Family," *IEEE International Solid State Circuits Conference*, February 1984, pp. 16-17.

[Hitc82] R.B. Hitchcock, Sr., "Timing Verification and the Timing Analysis Program," *Proc. 19th Design Automation Conference*, 1982, pp. 594-604.

[Joup87a] N.P. Jouppi, "Timing Analysis and Performance Improvement of MOS VLSI Designs," *IEEE Tran. Computer-Aided Design*, Vol. CAD-6, NO. 4, July 1987, pp. 650-665.

[Joup87b] N.P. Jouppi, "Derivation of Signal Flow Direction in MOS VLSI," *IEEE Tran. Computer-Aided Design*, Vol. CAD-6, No.3, May 1987, pp. 480-490.

[LiNi88] J. Liao and L.M. Ni, "A New CMOS Circuit Representation for Timing Verification," *Proc. 1988 International Symposium on Circuits and Systems*, June, 1988.

[LiNi89] J. Liao and L.M. Ni, "Boolean Behavior Extraction from Circuit Layout", *1989 International Symposium on VLSI Technology, Systems and Applications,* May, 1989, pp. 139-143.

[KoMc86] K.L. Kodandapani and E.J. McGrath, "A Wirelist Compare Program for Verifying VLSI Layouts," *IEEE Design & Test*, June 1986, pp. 46-51.

[MeCo80] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980.

[Nage75] L.A. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," *Tech. Rep. UCB ERL-M250, Univ. of California, Berkeley*, May 1975.

[OTOO86] G. Odawara, M. Tomita, O. Okuzawa, T. Ohta, and Z. Zhuang, "A Logic Verifier Based on Boolean Comparison," *Proceedings of 23rd Design Automation Conference*, June 1986, pp. 208-214.

[Oust85] J.K. Ousterhout, "A Switch-Level Timing Verifier for Digital MOS VLSI," *IEEE Tran. Computer-Aided Design*, Vol. CAD-4, NO. 3, July 1985, pp. 336-349.

[RaTr87] V.B. Rao, and T.N. Trick, "Network Partitioning and Ordering for MOS VLSI Circuits," *IEEE Tran. Computer-Aided Design*, Vol. CAD-6, no. 1, Jan. 1987, pp. 128-144.

[RoSh85] S.H. Robinson and J.P. Shen, "Towards a Switch Level Test Pattern Generation Program," *International Conference on Computer Aided Design*, 1985.

[RuPe83] J. Rubinstein, P. Penfield, Jr., and M.A. Horowitz, "Signal delay in RC tree networks," *IEEE Trans. Computer-Aided Design*, Vol. CAD-2, July 1983, pp. 202-211.

[VaDS85] A.V. Vasquez, S.W. Director, and K.A. Sakallah, "PRIMO: A VLSI Circuit Partitioner for Simulation Applications," *Proc. 1985 International Symposium on Circuits and Systems,"* 1985, pp. 1075-1078.

[Wata83] T. Watanabe et al., "A New Automatic Logic Interconnection Verification System for VLSI design," *IEEE Tran. CAD/ICAS*, Vol. CAD-2, No. 2, *April 1983, pp. 70-82.*

[WeEs85] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley, 1985.

[WeSa86] R. Wei and A.L. Sangiovanni-Vincentelli, "Proteus: a Logic Verification System for Combinational Circuits", *International Test Conference*, 1986, pp. 350-359.

[WuNW87] C.-F.E. Wu, L.M. Ni, and A.S. Wojcik, "Functional Recognition of Static CMOS Circuits," *Proc. 1987 International Conference on Computer Aided*

Design, Nov. 1987, pp. 306-308.